

# **Anais da Sessão de Ferramentas do SBCARS 2007**

**(SBCARS 2007)**

**30 de Agosto de 2007**

**Campinas, São Paulo, Brasil**

## **Promoção**

SBC – Sociedade Brasileira de Computação

## **Edição**

Cecília Mary Fischer Rubira (Instituto de Computação – Unicamp)

## **Organização**

Instituto de Computação – Unicamp

## **Realização**

Instituto de Computação – Unicamp

**Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software 2007: Campinas, SP.**

**Anais do sessão de ferramentas do SBCARS2007, Campinas, 30 de Agosto de 2007 / editor: Cecília Mary Fischer Rubira – Campinas (SP), Unicamp, 2007.**

**1. Engenharia de Software – Congressos. 2. Informática – Congressos. Rubira, Cecília Mary Fischer. SBCARS 2007, Campinas.**

Projeto gráfico: Maurício Bedo – Digital Assets, Campinas, SP.

Editoração:

Leonel Aguilar Gayard – Instituto de Computação, Unicamp

Cecília Mary Fischer Rubira – Instituto de Computação, Unicamp

**Esta obra foi impressa a partir de originais entregues, já compostos pelos autores**

## **Sumário / Contents**

### **Sessão de Ferramentas**

#### **Carga Dinâmica de Componentes via Biblioteca Brechó ..... 1**

Paula Fernandes (UFRJ)  
João Gustavo Prudêncio (UFRJ)  
Anderson Marinho (UFRJ)  
Marco Lopes (UFRJ),  
Leonardo Murta (UFRJ),  
Cláudia Werner (UFRJ)

#### **DA Manager®: Gerência e Avaliação do Reutilização de Ativos Digitais ..... 9**

Marcílio Oliveira (Laboratório de Inovação Digital Assets-Ci&T/Unicamp)  
Kleber Bacili (Digital Assets)  
José Cláudio Vahl Jr. (Laboratório de Inovação Digital Assets-Ci&T/Unicamp)

#### **GenArch: Uma Ferramenta baseada em Modelos para Derivação de Produtos de Software..... 17**

Elder Cirilo (PUC-Rio)  
Uirá Kulesza (PUC-Rio)  
Carlos Lucena (PUC-Rio)

#### **FATEsC - Uma Ferramenta de apoio ao teste estrutural de componentes... 25**

Vânia Teixeira (FGP/UNIVEM)  
Marcio Delamaro (UNIVEM)  
Auri M.R. Vincenzi (UNISANTOS)

#### **Uma Ferramenta para Configuração e Implantação de Sistemas Distribuídos de Tempo-Real Baseados em Componentes..... 33**

Sandro Andrade (Faculdade Ruy Barbosa)  
Cleber Ramos (Faculdade Ruy Barbosa)  
Aristoteles Marçal (Faculdade Ruy Barbosa)



## **Prefácio**

É com grande satisfação que incorporamos a Sessão de Ferramentas dentro do Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software de 2007 (SBCARS 2007).

Uma Sessão de Ferramentas tem o papel de um fórum para intercâmbio de experiências e de soluções desenvolvidas por universidades, centros de pesquisa e empresas, nas áreas de componentes, arquiteturas, reuso e outras áreas correlatas. Esse evento possibilita que a comunidade científica exponha seus trabalhos de forma prática e operacional e, assim, promova a transferência de tecnologia entre a academia e a indústria.

Esta é a primeira edição da Sessão de Ferramentas no SBCARS, que esperamos repetir nos próximos SBCARS. O comitê de avaliação, formado por destacados pesquisadores brasileiros, avaliou com cuidado as ferramentas, num processo à altura da importância da Sessão de Ferramentas. Assim, foram selecionadas cinco ferramentas, cujas apresentações orais e demonstrações ocorrerão no segundo dia do evento.

Considerando o caráter singular da Sessão de Ferramentas, que é justamente o de mobilizar a comunidade e promover a transferência de tecnologia no país, todos os esforços devem ser feitos para que seja mantida e melhorada a cada ano, incentivando-se, cada vez mais, a participação dos pesquisadores do nosso país.

Finalizando, gostaríamos de agradecer a todos que colaboraram para tornar possível essa Sessão de Ferramentas, principalmente aos autores dos trabalhos submetidos, aos membros do comitê de avaliação, aos revisores dos trabalhos, e particularmente a Valdirene Fontanette pelo seu apoio administrativo.

São Carlos, 8 de Agosto de 2007.

Antônio Francisco do Prado, UFSCar

Coordenador da I Sessão de Ferramentas do SBCARS

Valdirene Fontanette – UFSCar

Vice-coordenadora da I Sessão de Ferramentas do SBCARS



## Comissão Organizadora

- Cecília Mary Fischer Rubira (Coordenadora Geral) — Instituto de Computação (IC) - UNICAMP
- Antônio Francisco Prado e Valdirene Fontanette (Coordenadores da Sessão de Ferramentas) — Departamento de Computação (DC) - UFSCar
- Profa. Ariadne Rizzoni Carvalho — Instituto de Computação (IC) - UNICAMP
- Profa. Thelma Chiossi — Instituto de Computação (IC) - UNICAMP
- Patrick Henrique da Silva Brito — Instituto de Computação (IC) - UNICAMP
- Leonardo Pondian Tizzei — Instituto de Computação (IC) - UNICAMP
- Leonel Aguilar Gayard — Instituto de Computação (IC) - UNICAMP
- Ana Elisa de Campos Lobo — Instituto de Computação (IC) - UNICAMP
- Ivan Perez — Instituto de Computação (IC) - UNICAMP
- Claudia Regina da Silva — Instituto de Computação (IC) - UNICAMP
- Kleber Bacili — Digital Assets, Campinas,SP
- Ana Martini — Digital Assets, Campinas,SP
- Maurício Bedo — Digital Assets, Campinas,SP

## Comitê de Programa

- Cecília Mary Fischer Rubira (Coordenadora do Comitê de Programa) — Instituto de Computação (IC) - UNICAMP
- Cláudia Maria Lima Werner (Vice-Coodenadora do Comitê de Programa) — COPPE - UFRJ

## Membros do Comitê de Programa

- Alessandro Garcia — University of Lancaster
- Alexander Romanovsky — University of Newcastle
- Ana C.V. de Melo — USP
- Ana Paula Bacelo — PUCRS

- Antônio Francisco Prado — UFSCar
- Carlos Lucena — PUC-Rio
- Cecília M.F. Rubira — UNICAMP
- Cláudia Werner — UFRJ
- Cristina Gacek — University of Newcastle
- Eliane Martins — UNICAMP
- Glédson Elias — UFPB
- Guilherme Travassos — UFRJ
- Itana Gimenes — UEM
- Ivica Crnkovic — University of Mälardalen
- José Maldonado — USP-São Carlos
- Mehdi Jazayeri — University of Lugano
- Patrícia Machado — UFCG
- Paulo Borba — UFPE
- Paulo Masiero — USP-São Carlos
- Paulo Merson — Software Engineering Institute
- Regina Braga — UFJF
- Rogério de Lemos — University of Kent
- Rosana Braga — USP-São Carlos
- Sílvio Meira — UFPE
- Thaís Vasconcelos Batista — UFRN



## **Revisores**

- Ana Paula Bacelo — PUCRS
- André Domingues — SSC/ICMC-USP/São Carlos
- Antonio Francisco Prado — Universidade Federal de São Carlos
- Cecília Rubira — UNICAMP
- Eliane Martins — UNICAMP
- Elisa Nakagawa — Universidade de São Paulo - USP
- Gledson Elias — UFPB
- Helton Lima — Universidade Federal de Campina Grande
- Itana Maria de Souza Gimenes — Universidade Estadual de Maringá
- Leonardo Murta — COPPE/UFRJ
- Makelli Jucá — UFCG
- Masiero Paulo — Universidade de São Paulo/ICMC
- Regina Braga — Universidade Federal de Juiz de Fora
- Rosana Braga — ICMC-USP
- Thais Vasconcelos Batista — UFRN

## **Sociedade Brasileira de Computação**

### **Diretoria**

- Presidente: José Carlos Maldonado (ICMC - USP)
- Vice-Presidente: Virgílio Augusto Fernandes Almeida (UFMG)

### **Diretorias:**

- Administrativa: Carla Maria Dal Sasso Freitas (UFRGS)
- Finanças: Paulo Cesar Masiero (ICMC - USP)
- Eventos e Comissões Especiais: Marcelo Walter (UFPE)
- Educação: Edson Norberto Cáceres (UFMS)
- Publicações: Karin Breitman (PUC-Rio)
- Planejamento e Programas Especiais: Augusto Sampaio (UFPE)
- Secretarias Regionais: Aline dos Santos Andrade (UFBA)
- Divulgação e Marketing: Altigran Soares da Silva (UFAM)

### **Diretorias Extraordinárias:**

- Regulamentação da Profissão: Ricardo de Oliveira Anido (UNICAMP)
- Eventos Especiais: Carlos Eduardo Ferreira (USP)
- Cooperação com Sociedades Científicas: Taisy Silva Weber (UFRGS)

## **Conselho**

### **Mandato 2007-2011**

- Cláudia Maria Bauzer Medeiros (UNICAMP)
- Roberto da Silva Bigonha (UFMG)
- Cláudio Leonardo Lucchesi (UNICAMP)
- Daltro José Nunes (UFRGS)
- André Ponce de Leon F. de Carvalho (ICMC - USP)

### **Mandato 2005-2009**

- Ana Carolina Salgado (UFPE)
- Jaime Simão Sichman (USP)
- Daniel Schwabe (PUC-Rio)

### **Suplentes - Mandato 2007-2009**

- Vera Lúcia Strube de Lima (PUCRS)
- Raul Sidnei Wazlawick (UFSC)
- Ricardo Augusto da Luz Reis (UFRGS)
- Jacques Wainer (UNICAMP)
- Marta Lima de Queiroz Mattoso (UFRJ)



## Carga Dinâmica de Componentes via Biblioteca Brechó

**Paula Fernandes, João Gustavo Prudêncio, Anderson Marinho,  
Marco Lopes, Leonardo Murta, Cláudia Werner**

PESC/COPPE – Universidade Federal do Rio de Janeiro  
Caixa Postal 68.511 – 21945-970 – Rio de Janeiro – RJ – Brasil  
{paulacibele, gustavo, mlopes, murta, werner}@cos.ufrj.br,  
andymarinho@ufrj.br

***Resumo.** Este artigo apresenta uma ferramenta para carga dinâmica de componentes no ambiente Odyssey, utilizando a biblioteca de componentes Brechó. Ela foi implementada como forma de evoluir o mecanismo de carga dinâmica existente nesse ambiente. A Brechó foi utilizada para armazenar os componentes a serem instalados, possibilitando o desenvolvimento de uma solução mais flexível e organizada. O principal objetivo da ferramenta proposta é permitir que novas funcionalidades disponibilizadas através de componentes, armazenados na Brechó, possam ser adicionadas e removidas do ambiente Odyssey em tempo de execução, de acordo com a demanda dos usuários.*

### 1. Introdução

O ambiente Odyssey [8] tem como principal objetivo apoiar a reutilização de software por meio de técnicas de engenharia de domínio, linha de produtos e desenvolvimento baseado em componentes.

Durante a evolução deste ambiente, várias ferramentas de apoio às atividades de reutilização foram desenvolvidas. Porém, por serem fortemente acopladas ao Odyssey, afetavam negativamente características do ambiente como usabilidade, desempenho e evolução. Com a finalidade de contornar esse problema, foi realizada uma reengenharia do ambiente Odyssey, separando em um núcleo, denominado Odyssey Light, as funcionalidades identificadas como essenciais para a modelagem baseada em reutilização. As demais funcionalidades foram encapsuladas em ferramentas de apoio, denominadas *plug-ins*. Contudo, esta reengenharia demandou o desenvolvimento de um mecanismo que possibilitasse a carga dinâmica, por demanda, das ferramentas no núcleo [6], permitindo o gerenciamento de variabilidades em tempo de execução, por meio de seleção, recuperação e instalação de *plug-ins*.

No entanto, esse mecanismo de carga dinâmica possui algumas limitações. As ferramentas, suas dependências e sua localização são descritas em um arquivo XML. Para cada nova *release* do ambiente Odyssey, um novo arquivo deve ser criado apenas com as ferramentas compatíveis. Além disso, nenhuma ferramenta de apoio à edição desses arquivos foi desenvolvida, como forma de garantir sua consistência, por exemplo, em relação às ferramentas e suas dependências. Todos os arquivos binários correspondentes aos *plug-ins* e suas dependências são armazenados em um mesmo diretório em rede, não havendo nenhuma organização, dificultando sua manutenção. Outra limitação desta abordagem é o fato de determinado desenvolvedor não ter

informação sobre a utilização da sua ferramenta, uma vez que não há nenhum registro da instalação de *plug-ins* no Odyssey.

O objetivo deste artigo é apresentar uma evolução do mecanismo de carga dinâmica do ambiente Odyssey, que busca contornar os problemas citados anteriormente por meio da integração via Web Services deste ambiente com a biblioteca de componentes Brechó [1]. A Brechó é um sistema de informação para a Web que fornece mecanismos de documentação, armazenamento, busca e recuperação de componentes. Essa integração visa possibilitar que a Brechó faça a mediação entre o Odyssey e as suas ferramentas, carregadas dinamicamente.

Este artigo está organizado em cinco seções. Na Seção 2, são apresentados os principais conceitos envolvidos neste trabalho. Na Seção 3, é discutido o mecanismo proposto para variabilidade em tempo de execução por meio da carga dinâmica de componentes via Brechó. Na Seção 4, é apresentado um exemplo de utilização desse mecanismo. Por fim, na Seção 5, são destacadas as contribuições e trabalhos futuros.

## **2. Contextualização**

Nesta seção são discutidos alguns conceitos importantes para a contextualização deste trabalho. Na Seção 2.1 é abordado o conceito de variabilidade de sistema, que é uma das principais motivações para este trabalho. Na Seção 2.2 é apresentado o mecanismo de carga dinâmica anteriormente utilizado no ambiente Odyssey, que serviu como base para a abordagem proposta. Finalmente, a Seção 2.3 apresenta brevemente a Brechó.

### **2.1. Variabilidade de sistema**

Variabilidade de sistema é a habilidade do software ser eficientemente estendido, modificado, adaptado ou configurado para uso em um contexto particular [9]. Devido à relação custo-benefício, sistemas precisam ser liberados em diferentes distribuições (e.g. *standard* ou *professional*) [5]. Além disso, é importante permitir a configuração do sistema para se adequar às necessidades particulares dos clientes.

A variabilidade de sistema pode ocorrer em diferentes fases do ciclo de vida do software. Por exemplo, na fase de especificação, diferentes tecnologias permitem a representação da variabilidade do sistema, tais como o uso de elementos opcionais e variáveis em linhas de produtos para dar suporte à seleção de produtos [2] e à seleção de características em um modelo de domínio dentro de um processo de reutilização [7]. A variabilidade em tempo de implantação é uma das mais conhecidas, porém é menos flexível que a variabilidade em tempo de execução, visto que não é possível instalar novas funcionalidades durante a execução do software.

Além dessas abordagens voltadas para fases particulares do ciclo de vida do software, Hoek [4] e Gulp [3] sugerem o uso de arquiteturas de software para guiar a seleção de variabilidades a qualquer momento do ciclo de vida. Dessa forma, é possível definir variabilidade em tempo de projeto e aplicá-la no tempo de projeto, invocação ou execução. Em nosso caso, é importante permitir que a equipe de desenvolvimento do Odyssey descreva as variabilidades em tempo de desenvolvimento e que os engenheiros de software (usuários do Odyssey) selecionem as funcionalidades em tempo de execução. Alguns ambientes de desenvolvimento integrados, como Eclipse e NetBeans, utilizam uma abordagem baseada em *plug-ins* para atender a esse requisito.

## 2.2. Mecanismo anterior de carga dinâmica do Odyssey

Como mencionado na Seção 1, todas as ferramentas foram retiradas do núcleo do ambiente Odyssey e transformadas em *plug-ins*, tornando necessário o desenvolvimento de um mecanismo que possibilitasse a carga por demanda das ferramentas no núcleo.

Inicialmente, foi necessário o desenvolvimento de uma interface comum para todas as ferramentas, denominada *Tool*, que é acessada pelo ambiente Odyssey sempre que ele precisa consultar ou notificar os *plug-ins*. Essa interface define métodos que fornecem informações como, por exemplo, sobre quais menus devem ser incluídos no ambiente.

Nessa abordagem, uma ferramenta se torna um componente, ou *plug-in*, quando implementa a interface *Tool* e é empacotada em um arquivo JAR (*Java Archive*), armazenado em um diretório específico na Web. Todas as ferramentas, empacotadas como componentes, são descritas em um arquivo XML, contendo nome, tipo, descrição, localização no repositório e dependências.

Os tipos de componentes disponíveis são *kernel*, *plug-in* e *library*. Os componentes do tipo *kernel* estão no núcleo do ambiente Odyssey (por exemplo, o editor de diagramas). Os *plug-ins* podem ser ativados por meio de seleção, provendo variabilidade para o ambiente (por exemplo, adicionando a funcionalidade de geração de código por meio da ferramenta Odyssey-MDA-codegen). Finalmente, o tipo *library* engloba os componentes requeridos por outros componentes, mas que não representam ferramentas para o Odyssey (por exemplo, uma biblioteca específica de JDBC). Na Figura 1 temos a descrição da ferramenta Odyssey-XMI, utilizada para importação e exportação de modelos UML no formato XMI. Ela possui dependência para outros dois componentes do tipo *library*, que devem ser recuperados e instalados previamente, o que é feito de forma automática pelo mecanismo de carga dinâmica.

```
<component type="plugin" name="Odyssey-XMI-0.2.2.jar" description="Odyssey-XMI"
location="http://reuse.cos.ufrj.br/releases/components">
  <dependency name="MofRepository-1.0.0.jar"/>
  <dependency name="jmi-uml-1.4.jar"/>
</component>
```

Figura 1. Descrição da ferramenta Odyssey-XMI.

O mecanismo de carga dinâmica propriamente dito é responsável pela comunicação entre a instância do ambiente Odyssey e o diretório de componentes, incluindo a recuperação do arquivo descritor dos componentes e suas implementações, que ocorre através da utilização do protocolo HTTP. Após a fase de download, um novo *class loader* é criado para acessar as classes recuperadas. Então, a API de reflexão do Java é utilizada para acessar a classe declarada no arquivo *manifest* do componente. Essa classe é convertida para a interface *Tool* e colocada em uma coleção que contém todas as ferramentas instaladas.

## 2.3. A biblioteca Brechó

A biblioteca Brechó é um sistema de informação para Web que possui como principal objetivo fornecer mecanismos de documentação, armazenamento, busca, e recuperação de componentes [1].

Os mecanismos de publicação e documentação adotam um conceito flexível de componente, que vai além do arquivo binário, visando uma representação que inclua os possíveis artefatos produzidos durante o desenvolvimento do componente (como especificações, código fonte e manuais). A estrutura de documentação é fundamentada em categorias e formulários dinâmicos e configuráveis. Assim, é possível classificar um componente em várias categorias, além de criar e associar diferentes formulários de documentação a cada categoria, permitindo a construção da documentação do componente como um mosaico.

A organização interna da Brechó é dividida em níveis, que levam em consideração diferentes aspectos (cortes funcionais, temporais, etc.), para melhor representação do componente. O primeiro nível, denominado componente, representa conceitualmente as entidades armazenadas na Brechó, sem as informações concretas sobre as implementações dessas entidades. O nível distribuição representa um corte funcional sobre as entidades, fornecendo conjuntos de funcionalidades que são desejadas por grupos específicos de usuários. O nível *release* representa um corte temporal sobre as distribuições, no qual define versões dos artefatos que implementam as entidades em um determinado instante no tempo. A partir desse nível, as entidades passam a ter informações concretas sobre suas implementações, que usualmente existem em diferentes níveis de abstração (e.g. documentação do usuário, análise, projeto, código, binário). O nível pacote permite que seja feito um corte em níveis de abstração, possibilitando que sejam agrupados artefatos de acordo com o público alvo de reutilização. O nível licença possibilita a definição de níveis de serviço sobre os pacotes. Para cada pacote podem ser estabelecidas licenças específicas, que garantem regras entre os produtores e os consumidores dos componentes.

A Brechó oferece ainda outros mecanismos importantes, como, por exemplo, a representação da relação de dependência entre componentes, que é fundamental durante a recuperação de um componente, apresentando os componentes dos quais ele necessita para funcionar. Um outro mecanismo oferecido é o mapa de reutilização, que serve para auxiliar na atividade de identificação de responsabilidades de manutenção, mantendo as informações dos contratos firmados entre os produtores e consumidores, para cada componente reutilizado.

### **3. Carga dinâmica de componentes via Brechó**

Conforme apresentado na Seção 1, o mecanismo de carga dinâmica anteriormente presente no ambiente Odyssey possui algumas limitações. Grande parte desse mecanismo tem como base o arquivo XML com a descrição das ferramentas e suas dependências. Esse arquivo é gerado de forma estática para cada nova *release* do ambiente Odyssey sem nenhuma ferramenta de apoio, que ajude a garantir sua consistência. Todos os arquivos para instalação dos componentes são armazenados de forma aleatória em um diretório em rede, dificultando sua manutenção. Além disso, os desenvolvedores não possuem nenhum controle da utilização da sua ferramenta.

Como forma de superar essas limitações, o mecanismo proposto neste artigo para carga dinâmica de componentes no ambiente Odyssey utiliza algumas funcionalidades da Brechó. Para a integração Odyssey-Brechó optou-se pelo uso da tecnologia Web Services.



Foram criados na Brechó os serviços *UserService* e *ComponentService*, ilustrados na Figura 2, que disponibilizam parte das funcionalidades da biblioteca para uso por outras ferramentas. O serviço *UserService* possui métodos relacionados à autenticação dos usuários na Brechó, tornando possível a identificação dos consumidores dos componentes cadastrados. O serviço *ComponentService* engloba métodos relacionados à manipulação dos componentes, desde a obtenção da sua descrição até a sua recuperação, que são detalhados a seguir.

UserService
+login (user:String, password:String):String +logout (sessionId:String):void

ComponentService
+getAllComponents (releaseId:int):DataHandler +downloadReleasePkg (componentId:int, releaseId:int, pkgId:int, licenseId:int, sessionId:String):DataHandler +getReleaseId (componentName:String, releaseName:String):int

**Figura 2. Serviços *UserService* e *ComponentService*.**

O arquivo descritor dos componentes é gerado dinamicamente a partir da chamada ao método remoto *getAllComponents()*. Esse método possui como parâmetro o identificador da *release* do componente Odyssey, obtido por meio da chamada ao método remoto *getReleaseId()*, através da qual o mecanismo realizou a chamada remota. Ele retorna um arquivo XML com a descrição dos componentes e suas respectivas *releases* que dependem da *release* do componente Odyssey especificada. Além disso, esse arquivo contém a descrição das dependências diretas e indiretas desses componentes.

Devido à organização interna dos componentes na Brechó, foi necessário realizar algumas modificações em relação à forma como os componentes eram descritos no mecanismo anterior. A Figura 3 ilustra a nova descrição da ferramenta Odyssey-XMI.

```

<component id="2" name="Odyssey-XMI" description="Ferramenta Odyssey-XMI"
category="plugin">
  <release id="3" name="0.2.2">
    <package id="2" name="Default">
      <license id="1" name="Default" />
    </package>
    <dependency compid="3" relid="5" relname="1.0.0" pkgid="4" lcid="1" />
    <dependency compid="4" relid="4" relname="1.4" pkgid="3" lcid="1" />
  </release>
</component>

```

**Figura 3. Nova descrição da ferramenta Odyssey-XMI.**

O método *downloadReleasePkg()* possui como parâmetros os identificadores de componente, *release*, pacote e licença do pacote que deve ser recuperado, além do identificador do usuário para registro no mapa de reutilização.

Para que os componentes e suas dependências possam ser acessados e posteriormente instalados no ambiente Odyssey, eles devem ser previamente cadastrados na Brechó sob algumas diretrizes: (i) eles devem ser cadastrados nas categorias que identificam seu tipo (*kernel*, *library* ou *plugin*); (ii) suas *releases* devem possuir um artefato contendo o arquivo JAR que será utilizado no processo de instalação

no ambiente Odyssey e um pacote que contenha esse artefato. Além disso, é necessário estabelecer a relação de dependência entre as *releases* dos componentes do tipo *plug-in* com as *releases* do componente Odyssey cadastradas na Brechó com as quais são compatíveis. Essa é uma informação fundamental para o funcionamento do mecanismo proposto, pois ela é utilizada para identificar os componentes disponíveis para instalação em uma determinada *release* do componente Odyssey.

#### 4. Exemplo de utilização

A Figura 4 mostra o processo de disponibilização da *release* de um componente na Brechó. A Brechó permite a utilização de níveis do tipo padrão (*default*) para distribuição, pacote e licença, que facilitam esse processo.



Figura 4. Cadastro de um componente na biblioteca Brechó.

Inicialmente, é realizado o cadastro do componente e a escolha da sua categoria (Figura 4.a). Depois é realizado o cadastro da *release*, que passa a ter as informações concretas em relação à implementação do componente (Figura 4.b). Nessa etapa, é adicionado o arquivo JAR do componente. Por fim, é realizada a definição das dependências da *release* cadastrada (Figura 4.c).

No ambiente de configuração do Odyssey, o usuário pode definir a URL da biblioteca Brechó em que ele deseja acessar os componentes, o diretório local no qual esses componentes devem ser armazenados após a recuperação, além do usuário e senha utilizados para autenticação na Brechó (Figura 5.a). Após preencher esses dados, o usuário pode listar os componentes disponíveis e escolher se quer instalá-los ou desinstalá-los (Figura 5.b). Nessa etapa, algumas informações sobre os componentes podem ser visualizadas como, por exemplo, o tipo, o status e os componentes dos quais ele depende diretamente. No nosso exemplo, temos as informações da ferramenta Odyssey-XMI, que depende diretamente de outros dois componentes. Depois que a opção de instalação ou desinstalação é selecionada, o *wizard* de integração informa os componentes que serão instalados ou desinstalados, incluindo suas dependências diretas e indiretas (Figura 5.c). A ferramenta Odyssey-XMI possui dependência indireta para outros seis componentes, ou seja, para que suas dependências diretas possam ser

instaladas, outros componentes devem ser instalados concomitantemente. Finalmente, no caso da instalação, o *wizard* recupera os componentes e os carrega dinamicamente no ambiente, tornando a ferramenta acessível através do *menu* Tools. No caso da instalação da ferramenta Odyssey-XMI, o ambiente Odyssey passa a fornecer a funcionalidade de importação e exportação de modelos UML no formato XMI.

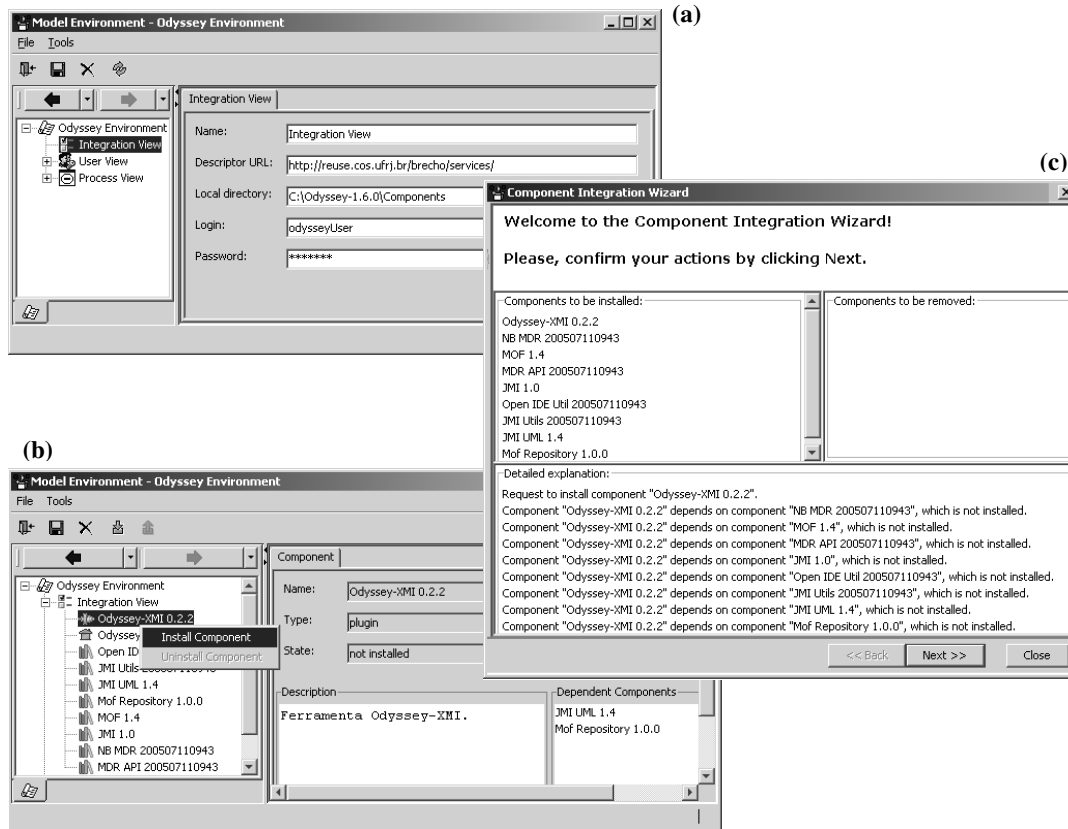


Figura 5. Carga dinâmica de um componente no ambiente Odyssey via Brechó.

## 5. Conclusões

Esse artigo apresentou uma ferramenta para carga dinâmica de componentes dentro de um ambiente em execução, utilizando uma biblioteca de componentes (i.e., Brechó). Essa ferramenta foi projetada e implementada para prover uma solução mais eficiente e flexível para o mecanismo de carga dinâmica do Odyssey [6]. Ela foi desenvolvida sem a necessidade de modificar a estrutura interna das ferramentas, utilizando como base parte dos mecanismos existentes. Além disso, permite um maior controle sobre a evolução dos componentes que podem ser instalados no Odyssey, facilitando o processo de geração de novas *releases* desse ambiente.

A carga dinâmica de componentes via Brechó trouxe benefícios tanto para os usuários do Odyssey quanto para os produtores das ferramentas. Para os primeiros, é possível acessar componentes sempre atualizados para serem utilizados de acordo com suas demandas em tempo de execução. Já para os produtores, o processo de disponibilização das suas ferramentas tornou-se mais simples e organizado, além de

permitir que o produtor tenha conhecimento sobre os seus consumidores, podendo notificá-los sobre, por exemplo, a liberação de novas versões.

Como trabalhos futuros, vislumbramos o tratamento da relação de conflito entre componentes, possibilitando a identificação de componentes incompatíveis e evitando o mau funcionamento dos componentes. Além disso, existe a possibilidade de tornar esse mecanismo de carga dinâmica sensível ao contexto, de forma que componentes possam ser automaticamente instalados sem a intervenção do usuário, com base nas ações do usuário dentro do ambiente Odyssey.

A Brechó pode ser acessada no link <http://reuse.cos.ufrj.br/brecho>. O ambiente Odyssey, na *release* 1.6.0, compatível com o novo mecanismo de carga dinâmica, pode ser obtido na própria Brechó.

### **Agradecimentos**

Os autores gostariam de agradecer à CAPES e ao CNPq pelo apoio financeiro, e à equipe de Reutilização da COPPE/UFRJ, que direta ou indiretamente apoiaram na realização desse trabalho.

### **Referências**

- [1] BRECHÓ (2007) "Projeto Brechó". In: <http://reuse.cos.ufrj.br/brecho>, acessado em 08/06/2007.
- [2] GARG, A., CRITCHLOW, M., CHEN, P., *et al.* (2003) "An Environment for Managing Evolving Product Line Architectures". In: *International Conference on Software Maintenance*, pp. 358-367, Amsterdam, Netherlands, September.
- [3] GURP, J., BOSCH, J., SVAHNBERG, M. (2001) "On the Notion of Variability in Software Product Lines". In: *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pp. 45-54, Amsterdam, Netherlands, August.
- [4] HOEK, A. (2004) "Design-Time Product Line Architectures for Any-Time Variability", *Science of Computer Programming*, v. 53, n. 3 (December), pp. 285-304.
- [5] LEON, A. (2000) *A Guide to Software Configuration Management*, Artech House Publishers.
- [6] MURTA, L., VASCONCELOS, A., BLOIS, A., *et al.* (2004) "Run-time Variability through Component Dynamic Loading". In: *XVIII Simpósio Brasileiro de Engenharia de Software, Sessão de Ferramentas*, pp. 67-72, Brasília, DF, Brasil, Outubro.
- [7] MILER, N., WERNER, C., BRAGA, R. (2000) "O uso de Modelos de Features na Engenharia de Aplicações". In: *IDEAS '00*, pp. 85-96, Cancun, México, April.
- [8] ODYSSEY (2007) "Projeto Odyssey". In: <http://reuse.cos.ufrj.br/odyssey>, acessado em 08/06/2007.
- [9] SVAHNBERG, M., GURP, J., BOSCH, J. (2005) "A taxonomy of variability realization techniques", *Software: Practice and Experience*, v. 35, n. 8 (April), pp. 705-754.

## **DA Manager®, gerência e avaliação da reutilização de ativos digitais.**

**Marcílio Oliveira<sup>1</sup>, Kleber Bacili<sup>2</sup>, José Cláudio Vahl Jr<sup>1</sup>**

<sup>1</sup>Laboratório de Inovação DigitalAssets-Ci&T/Unicamp  
Est. Telebrás, km 0,97 - UNICAMP - C.P.: 6123  
13081-970 – Campinas/SP - Brazil

<sup>2</sup>DigitalAssets  
Rodovia SP 340 Campinas/Mogi-Mirim, km 118,5 – Prédio 9C  
13086-902 – Campinas/SP - Brazil

{marcilio.oliveira, kleber.bacili, jose.vahl}@digitalassets.com.br

**Abstract.** *DA Manager is a digital assets management solution capable of delivering fast implementation and the promotion of software reuse initiatives within a company. This management tool supports SOA strategies and offers objective ways to measure results. DA Manager acts as a metadata repository, centralizing information about components, services and other reusable assets created during the application development cycle.*

**Resumo.** *O DA Manager é uma solução de gerenciamento de ativos digitais capaz de proporcionar a rápida implantação e promoção da iniciativa de reutilização de software na empresa, impulsionando a estratégia SOA através gerenciamento de serviços disponíveis e com formas objetivas de medir e acompanhar seus resultados. O DA Manager atua como um repositório de metadados, centralizando informações sobre componentes, serviços e outros ativos reutilizáveis criados ao longo do ciclo de desenvolvimento das aplicações.*

### **1. Introdução**

O desenvolvimento baseado em componentes é visto como uma forma de promover aumento de produtividade e qualidade na produção de software. No entanto, para sua aplicação efetiva, é necessário saber quais são e onde encontrar os ativos existentes, evitando retrabalho, capitalizando trabalhos anteriores e fazendo com que as soluções já desenvolvidas sejam imediatamente aplicadas em novos contextos.

O DA Manager® é uma ferramenta que visa promover a reutilização de software através do gerenciamento de ativos, com aplicação de métricas objetivas para avaliação de ganhos através da reutilização. O DA Manager® está fundamentado em tecnologias e padrões internacionais de modelagem e comunicação, como RAS (*Reusable Asset Specification*) [RAS 2006], SOA (*Service-Oriented Architecture*) e WebServices, visando prover interoperabilidade, padronização, distribuição e escalabilidade [Bacili 2006]. A ferramenta provê uma arquitetura Peer-to-Peer (P2P) para compartilhamento de componentes de software e um mecanismo de *Resource Discovery* na rede [Oliveira 2005] além de diversas funcionalidades de colaboração que tornam as equipes de

desenvolvimento de software mais produtivas na criação e reutilização de componentes e serviços de software.

## **2. Problema tratado**

A realidade atual do mundo corporativo exige uma grande agilidade em TI, não alcançada com processos tradicionais de desenvolvimento de software. É cada vez mais latente a necessidade de se desenvolver aplicações melhores e que atendam as necessidades do negócio em um tempo menor. Em um mercado altamente mutante, essa capacidade se traduz em importante diferencial competitivo.

No ciclo de desenvolvimento de software normalmente atuam diversos papéis. Por exemplo, analistas de requisitos, arquitetos, designers de interface, programadores, testers etc. A reutilização de software permeia diversas atividades do processo de desenvolvimento e boa parte dos papéis citados acima tem alguma interação com as ferramentas de promoção de práticas de reúso.

Pesquisas realizadas por Ambler [Ambler 2005] sugerem que o conhecimento de onde realmente existem ativos como aplicações, módulos de software, componentes e serviços é o principal ponto para promover a reutilização de software. O DA Manager® (figura 1) permite que cada pessoa envolvida no processo – do executivo ao desenvolvedor –, tenha uma visão clara do acervo digital com seus relacionamentos e interdependências, bem como das políticas que governam esses ativos e dos projetos que os produzem e/ou consomem.

Quando os envolvidos têm visibilidade dos componentes e serviços disponíveis para o uso, com informações sobre quando, onde e como eles devem (ou podem) ser usados, o acervo digital é consolidado, a redundância é eliminada e a complexidade, reduzida.

Além da visibilidade do acervo, todas as empresas prezam pelo controle dos investimentos frente a benefícios futuros almejados e, com rastreabilidade e avaliação de ganho através da aplicação de políticas de reutilização de ativos, é possível avaliar indicadores de ROI (*Return on Investment*) baseados em métricas objetivas. As métricas implementadas no DA Manager® foram apresentadas em publicações anteriores, como em [Oliveira 2006]. Mais detalhes sobre as formas de acompanhamento de ROI são apresentados na seção 4.3, de métricas objetivas de reutilização.



Figura 1. Tela principal da ferramenta.

### 3. Integrações e ficha técnica

A efetiva implantação de programas de reúso na prática é um processo importante que exige o envolvimento de diversas áreas em todo o ciclo de vida da produção de software. Esse processo precisa estar amparado por uma eficiente base tecnológica. A ferramenta de gestão de ativos digitais é uma das partes importantes, juntamente com outras ferramentas utilizadas ao longo desse processo.

O DA Manager® se encontra no centro dessa visão arquitetural, entre ferramentas e processo envolvendo reúso de software. Neste contexto, são envolvidas ferramentas de análise de código para extração de ativos, IDEs (Integrated Development Environment) de desenvolvimento, ferramentas de versionamento de código (Source Code Management) e barramentos de publicação de ativos em tempo de execução (ambientes de runtime).

A figura 2 ilustra uma visão geral deste conjunto de ferramentas posicionando o DA Manager® neste processo.

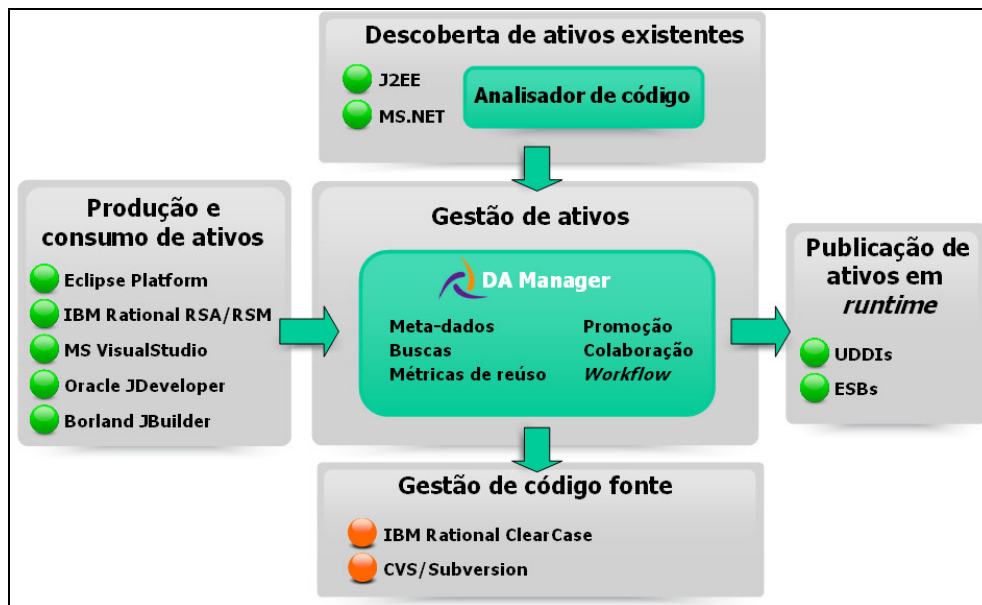


Figura 2. Visão geral da integração do DA Manager e outras ferramentas envolvidas em processo de reúso

A ficha técnica da ferramenta é apresentada na tabela a seguir:

Tabela 1. Informações técnicas da ferramenta.

<b>Ativos armazenados</b>	Qualquer tipo de ativos, independente de sua tecnologia, por exemplo: - Componentes J2EE, Microsoft, CORBA, etc - Rotinas COBOL / ADABA - WebServices - Procedimentos ABAP, etc - Stored Procedures
<b>Arquitetura</b>	- J2EE - XML e WebServices - RAS Compliant - SOA (Arquitetura Orientada a Services com API aberta e disponível)
<b>Servidores de Aplicação</b>	- IBM WebSphere - BEA WebLogic - JBoss - Oracle Application Server
<b>Servidores de Banco de Dados</b>	- IBM DB2 - Oracle - MS SQL Server
<b>Autenticação de usuários</b>	- Nativo do produto - Integrado com diretório LDAP (Microsoft Active Directory) - Extensões por meio de implementação de API
<b>Outras informações</b>	- Sistema Operacional - Multiplataforma (Windows, Linux, Unix, etc)



#### 4. Principais funcionalidades

O DA Manager® possui diversas funcionalidades para a viabilizar a implantação da estratégia de reúso. Valendo-nos de uma visão que leve em conta os objetivos estratégicos de atuação da ferramenta podemos classificar essas funcionalidades em: *Promoção dos ativos digitais, Gestão do acervo, Métricas objetivas de reúso e Formas de acelerar e facilitar a cultura do reúso de software.*

##### 4.1. Promoção dos ativos digitais:

- Indexação e buscas em todo o acervo (metadados e conteúdo de documentos);
- Mecanismos de colaboração como: notícias, RSS, fóruns de discussão, avaliação e feedback do uso ativos (*review*), etc.;
- Envio automático configurável de notificações por e-mail;
- Buscas remotas distribuídas em repositórios alternativos como SourceFORGE, Google e ComponentSource, usando mecanismo baseado na arquitetura P2P.

##### 4.2. Gestão do acervo:

- Uso de metadados com categorização flexível;
- Mapeamento das utilizações, relacionamentos e dependências entre os ativos;
- Gerência dos ativos e artefatos relacionados;
- Fácil configuração de perfis de acesso e parametrizações via interface;
- Análise de impacto com visualização gráfica das dependências.

##### 4.3. Métricas objetivas de reúso

As métricas disponíveis na ferramenta foram fundamentadas em referências científicas [Poulin 1997], e experiências de desenvolvimento baseado em componentes, para mensurar desvios e possíveis variáveis que influenciam no ganho com a reutilização de ativos, como overhead de administração, documentação, busca e integração. Algumas métricas de ganho, como RCA (*Reuse Cost Avoidance*) e ROA (*Return on Assets*) são baseados em pesquisas anteriores. As principais formas numéricas de acompanhamento do acervo são:

- Acompanhamento preciso de indicadores de economia relacionados ao reúso (RCA e ROA), conforme figura 3;
- Gráficos periódicos com resultados e estatísticas sobre o portfólio de ativos como: novos ativos criados, quantidade de visualizações, quantidade de downloads, buscas realizadas por palavra chave, etc. A figura 4 ilustra um desses gráficos.

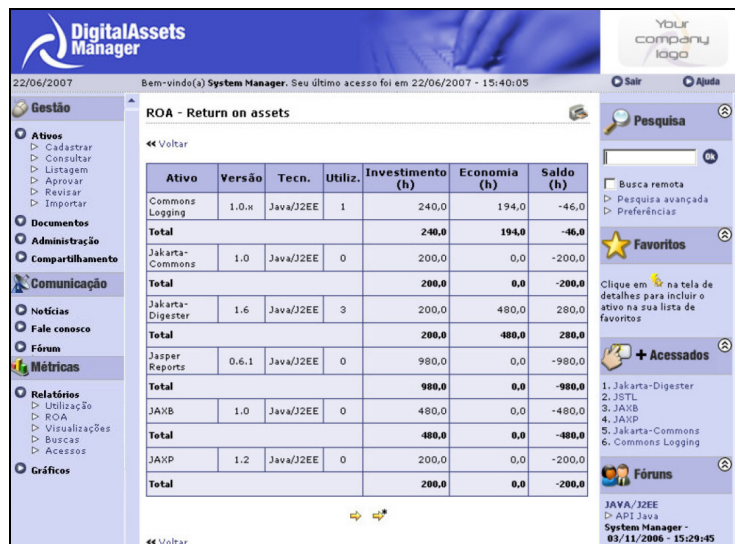


Figura 3. Métrica baseada em relatório ROA



Figura 4. Gráfico de visualização de ativos por mês

#### 4.4. Formas de acelerar o reúso:

A ferramenta conta com acervo inicial contendo ativos de código aberto (open-source) consagrados já cadastrados para impulsionar a estratégia de reúso (chamado de *Starting Library*), como por exemplo, alguns componentes disponíveis em sites da Jakarta, Hibernate.org, etc.

Como forma de fomento para a cultura de reúso, o DA Manager® se integra ao ambiente de desenvolvimento de várias formas:

- Plug-ins compatíveis com Eclipse (figura 5), JDeveloper e Microsoft VisualStudio;

- Integração com os repositórios de controle de versão de código-fonte, como o CVS, o Subversion e o IBM Rational ClearCase;
- Agente Windows® para acesso rápido (aplicação “in tray”).

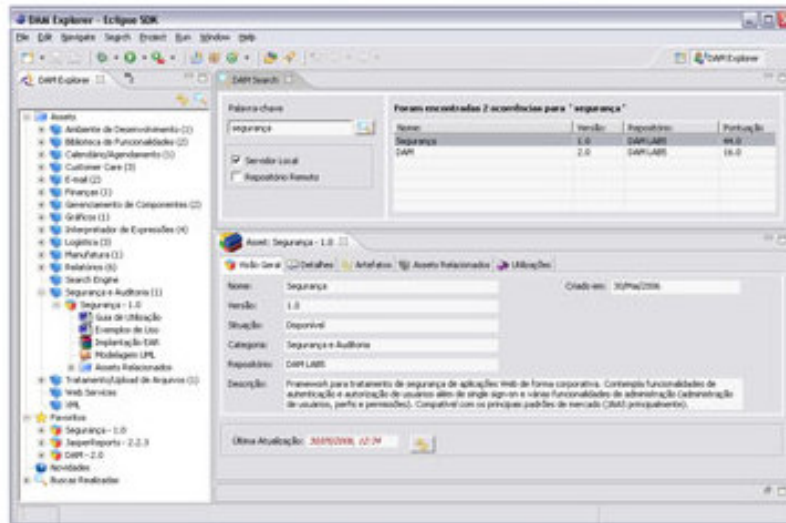


Figura 5. Tela principal do plug-in eclipse integrado ao repositório.

## 5. Acesso para demonstração

É possível visualizar a ferramenta em ação acessando o site da DigitalAssets na web ([www.digitalassets.com.br](http://www.digitalassets.com.br)) e solicitando uma demonstração guiada pelos nossos consultores.

## Referências

- Ambler, Scott W. Nalbhone, John. Vizidos, Micheal J. “The Enterprise Unified Process: Extending the Rational Unified Process”. Printice Hall, 2005.
- Bacili, K., Oliveira, M., “DigitalAssets Manager: sharing and managing software development assets”, OOPSLA'06 Demo Session, ACM, NY, 2006, pp. 700-701.
- Oliveira, M. e Bacili, K. “O reuso na prática - O reuso como diferencial competitivo em produtividade e qualidade no desenvolvimento de software.” Revista Mundo Java, edição de julho/2006 (nº. 18).
- Oliveira, M., Garcia, I., Nunes, A. (2005). “RCCS: uma Rede de Compartilhamento de Componentes de Software”, Brazilian Symposium of Computers Networks (SBRC), Fortaleza, Brazil, 2005.
- Poulin, Jeffrey. “Measuring Software Reuse: Principles, Practices, and Economic Models”. Addison-Wesley, Reading, MA, 1997.
- RAS, “Reusable Asset Specification, OMG Available Specification”. Object Management Group, (2006). Available at: <http://www.omg.org/technology/documents/formal/ras.htm>. Last view: 2007-03.



## GenArch: Uma Ferramenta baseada em Modelos para Derivação de Produtos de Software

Elder Cirilo<sup>1</sup>, Uirá Kulesza<sup>1</sup>, Carlos José Pereira de Lucena<sup>1</sup>

<sup>1</sup>Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro (PUC-RIO) - Rio de Janeiro – RJ – Brazil

{ecirilo,uira,lucena}@inf.puc-rio.br

***Resumo.** Este artigo apresenta uma ferramenta baseada em modelos para derivação de produto, denominada GenArch. A ferramenta é centrada na definição de três modelos – característica, arquitetura e configuração, os quais permitem a instanciação automática de linhas de produto de software ou frameworks. A ferramenta foi implementada na forma de um plug-in da plataforma Eclipse e utiliza a tecnologia EMF para manipulação de seus modelos. Anotações Java foram também definidas para permitir a geração de versões iniciais dos modelos a partir do código de implementação da arquitetura da linha de produto.*

### 1. Introdução

Pesquisas atuais e experiências práticas [3, 6, 12] sugerem que para se obter um progresso significativo com respeito ao reuso de software é necessário focalizar a modelagem e desenvolvimento de linhas de produtos ao invés de se desenvolver sistemas individualmente. A engenharia de linha de produtos procura explorar os pontos comuns e variáveis existentes entre sistemas de uma maneira sistemática. Através de uma linha de produtos, diferentes customizações de aplicações de um mesmo domínio podem ser rapidamente criadas a partir de um conjunto de artefatos reusáveis (arquitetura comum, componentes, framework, modelos, etc).

De acordo com Krueger [7], a adoção de uma abordagem de desenvolvimento de linha de produtos é mais difícil que simplesmente construir um sistema. Em parte porque durante várias décadas foram desenvolvidas métodos, técnicas e ferramentas centrados principalmente na construção de sistemas de software únicos, havendo poucas ferramentas para o gerenciamento e manipulação de variações de linha de produtos de software. Recentemente várias pesquisas e experiências industriais tem motivado o desenvolvimento de abordagens que lidam explicitamente com o gerenciamento de variabilidades de linhas de produtos. Algumas ferramentas para gerência de variabilidades e derivação automática de produtos de software têm sido desenvolvidas, tais como, Gears e pure::variants. Essas ferramentas apresentam funcionalidades úteis para a derivação de produtos de software, mas são, em geral, complexas e pesadas para serem usadas pela comunidade de desenvolvedores com pouca familiaridade com vários dos novos conceitos propostas pela comunidade de linhas de produto de software.

Nesse contexto, esse artigo apresenta a ferramenta GenArch que busca auxiliar engenheiros de software nas atividades de adoção e derivação de linhas de produto de software. A derivação automática de produtos na ferramenta é realizada através da definição de três modelos: característica, configuração e arquitetura. Versões iniciais desses modelos são geradas automaticamente pela ferramenta a partir da anotação do código de implementação da arquitetura de linha de produto.

Este artigo está organizado da seguinte forma: a Seção 2 apresenta uma visão geral da ferramenta, apresentando seu funcionamento geral e sua arquitetura baseada em tecnologias Eclipse. A Seção 3 descreve brevemente a preparação do framework JUnit para ser instanciado automaticamente pela GenArch. Finalmente, a seção 4 apresenta as conclusões do trabalho e direções para pesquisas futuras.

## 2. GenArch – Generative Architecture Plugin

GenArch é uma ferramenta baseada em modelos que visa simplificar a derivação automática [13] de novos membros de linha de produtos de software (LPS) ou instâncias de frameworks OO. A ferramenta é baseada nos conceitos de Programação Generativa (PG) [4]. PG contempla métodos e técnicas que permitem a geração automática de membros de uma linha de produtos a partir de especificações de alto nível (ex: modelos). Ela motiva a separação do espaço de problema do espaço de solução. O espaço de problema representa conceitos, abstrações e características (features) existentes no domínio de uma linha de produto. O espaço de solução agrega a arquitetura de software e respectivos componentes de implementação que são usados na geração de um membro da família de sistemas. O mapeamento entre os espaços de problema e solução é definido em programação generativa, através do conhecimento de configuração. O conhecimento de configuração define como combinações específicas de características no espaço de problema são mapeadas para uma configuração específica de componentes no espaço de solução.

A ferramenta GenArch propõe a definição de três modelos os quais são usados para representar as variabilidades e elementos de implementação de uma LPS, são eles: (i) modelo de arquitetura; (ii) modelo de característica; e (iii) modelo de configuração. Cada modelo representa: o espaço de solução, espaço de problema e conhecimento de configuração da organização proposta por Programação Generativa. A seguir cada modelo é brevemente descrito.

O modelo de arquitetura (espaço de solução) define uma representação visual dos elementos de implementação da LPS, tais como, classes, aspectos, templates e arquivos de configuração ou figuras. Tais elementos são organizados em diferentes componentes que definem a arquitetura do sistema. O GenArch permite a geração automática da versão inicial do modelo de arquitetura através do *parsing* dos diretórios contendo os elementos de implementação da arquitetura de LPS. O modelo de arquitetura é criado de forma a permitir relacionar os elementos de implementação com as variabilidades representadas no modelo de característica.

O modelo de característica (espaço de problema) é usado no GenArch para representar as variabilidades da LPS. A ferramenta adota o modelo de característica proposto por Czarnecki et al [4], o qual permite modelar características obrigatórias, opcionais e alternativas, com suas respectivas cardinalidades, restrições (*constraints*) e atributos. O plugin FMP (*Feature Modeling Plugin*) [1] é usado para especificar o modelo de característica.

Finalmente, o modelo de configuração (conhecimento de configuração) é responsável por definir o mapeamento entre características e elementos de implementação. Um conjunto de relações de dependência entre as características (espaço de problema) e os elementos de implementação (espaço de solução) são definidas, de forma a decidir quais elementos de implementação devem ser instanciados

a partir do fornecimento de uma instância do modelo de característica (um produto), durante o processo de derivação de produto suportado automaticamente pelo GenArch.

## 2.1. Visão Geral de Funcionamento

A ferramenta GenArch é baseada em modelos, sendo fundamental sua especificação para permitir a derivação automática da linha de produto. Versões iniciais dos modelos são geradas automaticamente a partir do *parsing* do diretório contendo elementos de implementação de um projeto Java. Alguns desses elementos podem também conter anotações GenArch específicas que são consideradas na geração dos modelos.

A Figura 1 apresenta uma visão geral do funcionamento da ferramenta GenArch, assim como ilustra as tecnologias usadas no seu desenvolvimento. Inicialmente ocorre a preparação do código dos elementos de implementação para permitir a geração inicial dos modelos (passo 1). Nessa etapa o engenheiro de domínio, usa um conjunto explícito de anotações para indicar no código dos elementos de implementação da linha de produto: (i) quais elementos corresponde à implementação específica das características variáveis (variabilidades) da linha de produto (anotação do tipo @Feature); e (ii) quais elementos correspondem a pontos de extensão (*hotspots*) da linha de produto (anotação do tipo @Variability).

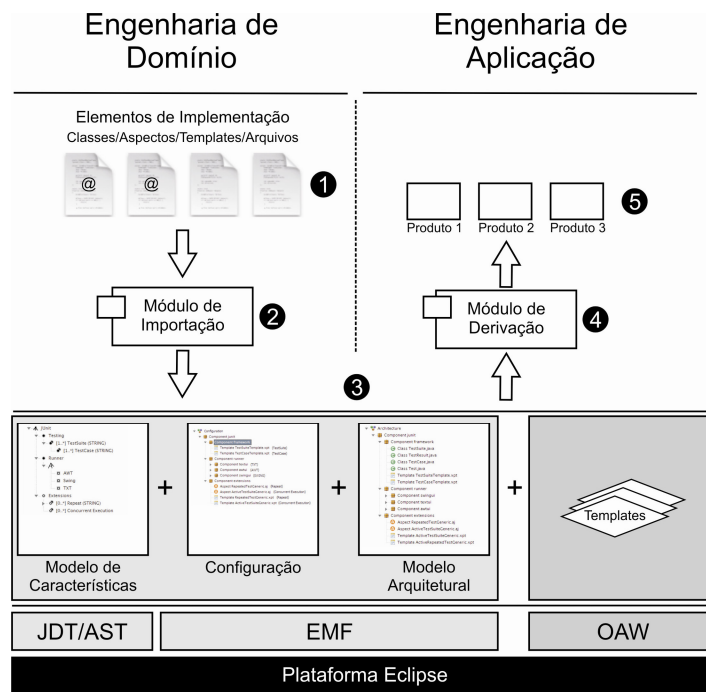


Figura 1. Arquitetura da ferramenta GenArch

Em seguida, o módulo de importação da ferramenta GenArch pode realizar um *parsing* no diretório contendo os elementos de implementação com suas respectivas anotações e gera uma versão inicial do modelo (passo 2) de arquitetura, de característica, de configuração e dos templates. A geração dos modelos de característica e configuração é baseada nas anotações (criadas no passo 1) que indicam as implementações específicas de características. Os templates são criados a partir das anotações que indicam quais são os pontos de extensão da linha de produto. O módulo

de importação pode ser acionado a qualquer momento. No início da iteração quando se deseja importar os modelos a partir de código existente ou em um momento qualquer, por exemplo, quando novos requisitos foram implementados na linha de produtos.

Após essa geração inicial, o engenheiro de domínio deve refinar e incrementar as versões iniciais dos modelos gerados (passo 3). As seguintes modificações são tipicamente realizadas: (i) adição de novas características ou reorganização das existentes; (ii) adição de novas relações de dependência entre características e elementos de implementação no modelo de configuração, sobretudo aquelas que não foram possíveis de serem especificadas através de anotações; e (iii) refinamento da implementação dos templates para contemplar a customização de alguma variabilidade a partir de informação coletada pelo modelo de característica.

Após o refinamento de cada um dos modelos, o módulo de derivação da ferramenta GenArch permite a instanciação automática de membros da linha de produto (passo 4). Essa derivação demanda inicialmente a especificação de uma instância do modelo de característica que representa aquele produto. Ou seja, o engenheiro de aplicação realiza a resolução de características, escolhendo um conjunto específico de elementos de implementação que satisfaça o produto que ele deseja. A ferramenta GenArch usa essa instância do modelo de característica, o modelo de configuração e o modelo de arquitetura para então gerar o produto desejado pelo usuário.

O módulo de derivação varre todo o modelo de arquitetura e para cada elemento ele verifica se existe uma relação de dependência no modelo de configuração entre tal elemento e alguma característica. Caso exista tal relação de dependência (ou seja, o elemento só deve ser instanciado caso uma dada característica tenha sido selecionada), a instância do modelo de características é consultada e dependendo da configuração da característica o elemento é adicionado ao produto. Se não existir uma relação de dependência o elemento de implementação será adicionado ao produto, representando nesse caso uma característica obrigatória. Esse processo de geração demanda a criação de um projeto Java/Eclipse (passo 5) contendo todos os elementos de implementação que implementam as características selecionadas, assim como o processamento de templates para geração de classes e/ou aspectos específicos que representam instâncias concretas dos pontos de extensão.

## 2.2. Arquitetura da Ferramenta

A Figura 1 também apresenta uma visão geral da arquitetura de implementação do GenArch. A ferramenta foi implementada como um plug-in da plataforma Eclipse utilizando vários *kits* de desenvolvimento orientado a modelos disponíveis em tal plataforma. Todos os modelos do GenArch foram construídos e são manipulados usando os recursos disponibilizados pelo *Eclipse Modeling Framework* (EMF) [2]. EMF é um framework Java/XML que permite a construção de ferramentas orientadas a modelos. Ele possibilita a geração de classes Java para criação e edição de modelos baseadas num dado meta-modelo especificado em XML Schema ou usando diagramas de classe UML. O modelo de característica usado na ferramenta GenArch é especificado pelo plug-in FMP (*Feature Modelling Plug-in*). Os modelos criados no FMP são também modelos baseados no EMF.

A ferramenta GenArch utiliza o plug-in openArchitectureWare (oAW) [11] para especificar seus templates. A linguagem XPand que faz parte do oAW é usada com tal finalidade. XPand é uma linguagem simple e expressiva. Permitindo a especificação de



templates que podem ser customizados usando informações provenientes do modelo de característica.

### 3. Exemplo Ilustrativo: Framework JUnit

Nesta seção utilizamos o framework JUnit para ilustrar como a ferramenta Genarch pode ser utilizada: (i) na preparação da linha de produtos a partir de código já existente; e (ii) para derivar membros dessa linha de produtos. O propósito principal do framework JUnit é permitir o projeto, implementação e execução de testes de unidade em aplicações Java. A Figura 2 apresenta os principais elementos da arquitetura do JUnit. As suas principais funcionalidades são: a definição de casos de testes ou suítes para serem executadas (*TestCase* e *TestSuite*); a execução de casos de testes ou suítes (*BaseTestRunner* e *TestRunner*); e coleta (*TestResult*) e apresentação visual dos resultados. Além disso, diferentes extensões podem ser implementadas para adicionar novas funcionalidades dentro do núcleo do framework JUnit. Alguns exemplos de extensões simples e que foram implementadas são: (i) permitir ao JUnit executar suítes de teste em threads separadas (*ActiveTestSuite*) e esperar pelo término de todos os teste. Na implementação dessa extensão é necessário observar os eventos que ocorrem quando um caso de teste inicia sua execução, o evento que ocorre quando cada método de teste roda e o evento que ocorre quando o caso de teste pára de rodar; (ii) e permitir o JUnit executar cada método de teste repetidamente (*RepeatedTestGeneric*). Na implementação dessa extensão é necessário observar o início e o término da execução de cada método de teste. Nosso estudo de caso, utilizou uma versão do JUnit que considerou a implementação dessas extensões usando programação orientada a aspectos. Detalhes adicionais sobre tal estudo de caso podem ser encontrados em [10].

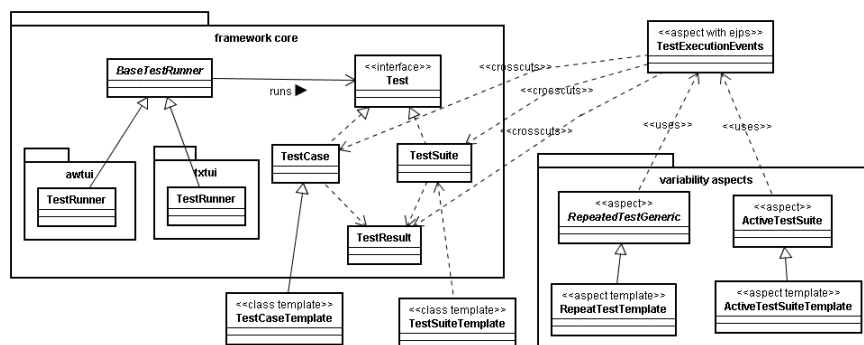


Figura 2. Arquitetura Orientada a Aspectos do JUnit

#### 3.1. Criando Anotações em Classes e Aspectos

O primeiro passo do nosso estudo de caso foi criar anotações GenArch, indicando explicitamente no código-fonte do framework JUnit, relacionamentos entre características e elementos de implementação, assim como suas variabilidades. No JUnit, as classes *TestCase* e *TestSuite* além de serem pontos de extensão (*hotspots*) também estão relacionadas com as características *Test Case* e *Test Suite*. Dessa forma, estas classes foram anotadas com `@Feature` para descrever o mapeamento e `@Variability` para indicar a criação de um template para cada classe. O código anotado pode ser visto na Figura 3. Além dessas classes, os aspectos

RepeatedTestGeneric e ActiveTestGeneric também foram anotados com @Feature e @Annotation, pois representam aspectos abstratos que são pontos de extensão orientados a aspectos do framework JUnit.

```
@Feature(name="TestCase",parent="TestSuite",type=FeatureType.mandatory)
@Variability(type=VariabilityType.hotSpot,feature="TestCase")

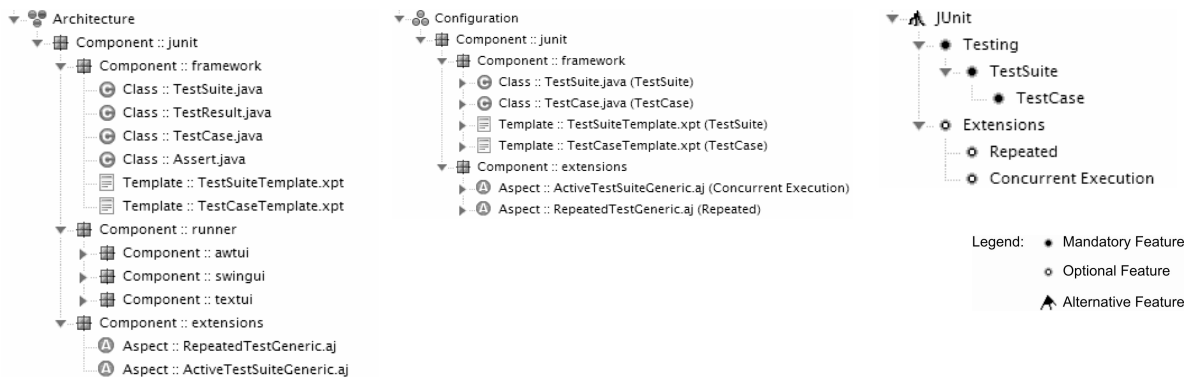
public abstract class TestCase extends Assert implements Test {

    private String fName;
    public TestCase() {
        fName= null;
    }
    public TestCase(String name) {
        fName= name;
    }
    ...
}
```

Figura 3. Código parcial da classe TestCase anotada.

### 3.2. Gerando e Atualizando os Modelos

O passo seguinte do estudo de caso foi adicionar ao projeto Java/Eclipse em uso a natureza de um projeto GenArch. Essa operação é feita através da opção “Convert to Genarch project”. Antes do início da importação o plug-in solicita ao engenheiro de domínio que especifique os *source folders* que serão explorados. O módulo de importação é então acionado e a partir do conteúdo dos *source folders* e das anotações descritas no código a ferramenta gera uma versão inicial de cada um dos modelos e dos templates<sup>1</sup>. As versões iniciais dos modelos gerados para o JUnit são apresentadas na Figura 4.



(a) Modelo de Arquitetura

(b) Modelo de Configuração

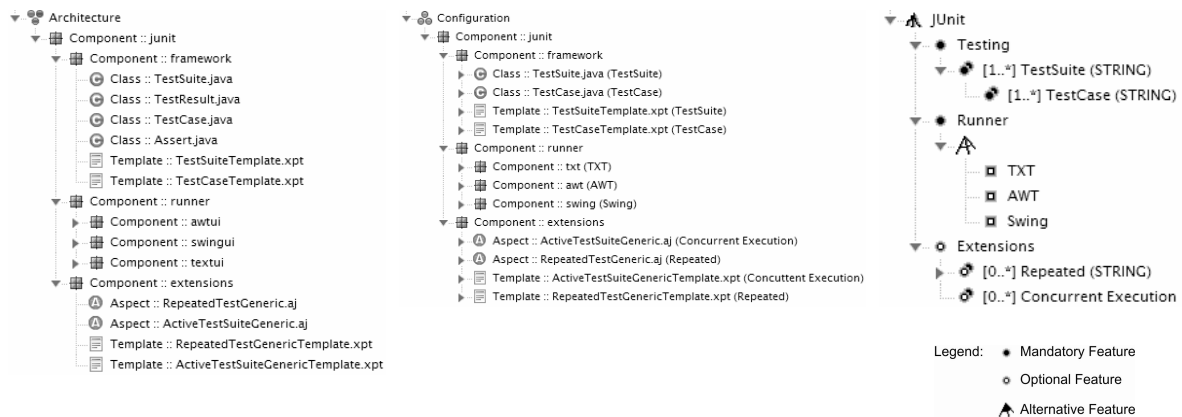
(c) Modelo de Características

Figura 4. JUnit GenArch Models – Versão inicial.

O modelo de arquitetura da Figura 4(a) contém todos os elementos que implementam o framework JUnit. O modelo de configuração, apresentado na Figura

<sup>1</sup> A geração do código inicial dos templates no GenArch é realizada através do processamento das anotações do tipo @Variability.

4(b) possui apenas as configurações que foram representadas no código, através das anotações `@Feature` e `@Variability`. Para o estudo de caso do JUnit foram criadas relações de dependência no modelo de configuração entre as classes `TestCase` e `TestSuite` se relacionando com as características `TestCase` e `TestSuite`, respectivamente. Também os templates `TestCaseTemplate` e `TestSuiteTemplate` que foram gerados a partir das anotações `@Variability` das classes `TestCase` e `TestSuite`, respectivamente, são relacionados com as mesmas características. Os aspectos `ActiveTestSuiteGeneric` e `RepeatedTestGeneric` foram relacionados com as características `ConcurrentExecution` e `Repeated`, respectivamente. As características que foram descritas nas classes e no aspectos também possibilitaram a criação de uma versão inicial do modelo de características, mostrada na Figura 4(c).



(a) Modelo de Arquitetura

(b) Modelo de Configuração

(c) Modelo de Características

**Figura 5. Versão final dos modelos de características e configuração**

Como mencionado anteriormente (Seção 2.1), as versões apresentadas na Figura 4 são apenas as versões iniciais dos modelos. Foi necessário o incremento e refinamento dos modelos de configuração e características para tornar possível a derivação automática de instâncias do framework JUnit. A Figura 5 apresenta a versão final dos modelos. Neste caso os componentes que implementam a interface de execução dos testes não foram configurados inicialmente pela ferramenta e tiveram de ser configurados manualmente. No modelo de características, Figura 5(c), as características Runner, TXT, AWT e Swing foram adicionadas. Cada uma dessas características representa um tipo de interface de execução dos testes do JUnit. Já no modelo de configuração, apresentado na Figura 5(b), foram criados relacionamentos entre os componentes (pacotes Java) que implementam cada uma das interfaces de execução e suas respectivas características (TXT, AWT e Swing). Ainda no modelo de configuração os templates `ActiveTestSuiteGenericTemplate` e `RepeatedTestGenericTemplate` foram associados com as características `ConcurrentExecution` e `Repeated`. Esses templates representam subaspectos dos aspectos abstratos existentes no mesmo componente.

### 3.3. Derivando um Membro da Linha de Produto ou Framework

Na última etapa da ferramenta GenArch, um membro da linha de produtos é derivado. A derivação se dá a partir da criação de uma instância do modelo de características, dos

mapeamentos descritos no modelo de configuração e do modelo de arquitetura. O primeiro passo nessa etapa é a seleção e configuração das variabilidades no modelo de características usando o plug-in FMP. Em seguida, o engenheiro de aplicação passa para a ferramenta GenArch a instância do modelo de característica criado, assim como o modelo de configuração e de arquitetura da linha de produto sendo considerada. A partir desses modelos a ferramenta seleciona quais elementos são necessários para implementar o produto a ser gerado. Esses elementos são copiados para suas respectivas pastas em um novo projeto Eclipse, que é especificado pelo engenheiro.

#### 4. Conclusões e Trabalhos Futuros

Este artigo apresentou uma visão geral do GenArch, uma ferramenta baseada em modelos que em combinação com anotações Java permite a derivação automática de membros de uma linha de produtos. A ferramenta é implementada utilizando a plataforma Eclipse, assim como diversas tecnologias de desenvolvimento orientado a modelo disponíveis em tal plataforma. Os recursos da ferramenta foram ilustrados usando o framework JUnit, detalhes adicionais desse e outros estudos de caso podem ser encontrados em [5]. Atualmente, diversas funcionalidades estão sendo incorporadas a ferramenta, entre elas: (i) sincronização entre os modelos, código e anotações; (ii) customização de pontos de corte de aspectos usando o modelo generativo orientado a aspectos proposto em [9,8]. Finalmente, novos estudos com linhas de produto mais complexas estão sendo planejados para validar o uso da ferramenta em cenários reais.

#### Referências

- [1] M. Antkiewicz, K. Czarnecki. FeaturePlugin: Feature modeling plug-in for Eclipse, OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, 2004.
- [2] F. Budinsky, et al. Eclipse Modeling Framework. Addison-Wesley, 2004.
- [3] P. Clements, L. Northrop. Software Product Lines: Practices and Patterns. 2001: Addison-Wesley Professional.
- [4] K. Czarnecki, U. Eisenecker. Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
- [5] GenArch – Generative Architectures Plugin, URL: <http://www.teccomm.les.inf.puc-rio.br/genarch/>.
- [6] J. Greenfield, K. Short. Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools. 2005: John Wiley and Sons.
- [7] C. Krueger. “Easing the Transition to Software Mass Customization”. In Proceedings of the 4th International Workshop on Software Product-Family Engineering, pp. 282–293, 2001.
- [8] U. Kulesza, et al. Mapping Features to Aspects: A Model-Based Generative Approach. Early Aspects, AOSD'2007, Vancouver, Canada. LNCS 4765, Springer-Verlag.
- [9] U. Kulesza, C. Lucena, P. Alencar, A. Garcia. Customizing Aspect-Oriented Variabilites Using Generative Techniques. International Conference on Software Engineering and Knowledge Engineering (SEKE'06). July 2006. San Francisco.
- [10] U. Kulesza, R. Coelho, V. Alves, A. C. Neto, A. Garcia, C. Lucena, C.; A. V. Staa, P. Borba. Implementing Framework Crosscutting Extensions with XPIs and AspectJ. XX Simpósio Brasileiro de Engenharia de Software (SBES'2006). 2006. Florianópolis.
- [11] openArchitectureWare, URL: <http://www.eclipse.org/gmt/oaw/>
- [12] D. Weiss, C. Lai. Software Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley Professional, 1999.
- [13] S. Deelstra, M. Sinnema, J. Bosch. Product derivation in software product families: a case study. Journal of Systems and Software 74(2): 173-194, 2005.

## FATEsC - Uma Ferramenta de apoio ao teste estrutural de componentes

Vânia Somaio Teixeira<sup>1,2</sup>, Marcio Eduardo Delamaro<sup>1</sup>,  
Auri Marcelo Rizzo Vincenzi<sup>3</sup>

<sup>1</sup>Programa de Pós-graduação em Ciência da Computação (PPGCC)  
Centro Universitário Eurípides de Marília (UNIVEM) – Marília, SP

<sup>2</sup>Faculdade Gennari&Peartree (FGP) – Pederneiras, SP

vania@fgp.com.br, delamaro@pesquisador.cnpq.br

<sup>3</sup>Universidade Católica de Santos (UNISANTOS) – Santos, SP

auri@unisantos.br

**Abstract.** *The use of components in software development brought many benefits, but also problems for the testing activity, which can be seen under two perspectives: the developer's and the user's. The problem in both of them is the lack of information; for the developer who doesn't know all the contexts in which the component will be used and for the user who doesn't know details about the component development or test. The FATEsC Tool allows the developer to generate testing metadata and to associate them to the component, permitting the user of the component to manipulate such information in order to evaluate her or his own test set, considering the coverage of the component's code.*

**Resumo.** *O uso de componentes em desenvolvimento de software trouxe muitos benefícios, mas também problemas para a atividade de teste, que pode ser vista sob duas perspectivas: a do desenvolvedor e a do usuário do componente. O problema, em ambas, é a falta de informação, do desenvolvedor por não conhecer todos os contextos de utilização do componente e do usuário pela falta de documentação adequada sobre o componente e sobre seu desenvolvimento e teste. A Ferramenta FATEsC permite que o desenvolvedor gere metadados de teste e os associe ao componente, viabilizando ao usuário do componente consultar tais informações para avaliar seus conjuntos de teste, levando em conta também, a cobertura do código do componente.*

### 1. Introdução

O objetivo primordial da Engenharia de Software é a construção de software com qualidade e com baixo custo. Para tanto, diversos métodos, técnicas e ferramentas têm sido propostos e utilizados buscando o aumento da produtividade no desenvolvimento de software. Uma das formas para tal foi o reaproveitamento de artefatos de software, assim, em vez de iniciar-se sempre um projeto de software “do zero”, são reutilizadas soluções e experiências adquiridas em projetos anteriores. É o que se costuma chamar de reúso de software. Uma forma de reúso de software, embora não seja uma prática

recente, é o desenvolvimento baseado em componentes, que é a reutilização de software por meio do encapsulamento de funcionalidades em unidades que são independentes e podem ser acopladas a um sistema por meio de uma interface conhecida.

Beydeda e Gruhn (2003) afirmam que os desenvolvimentos baseados em componentes tendem a trazer grandes benefícios na produção de software, mas, por outro lado, adicionam dificuldades que precisam ser transpostas, principalmente na fase de testes. Dificuldades essas, como a falta de informação sentida tanto pelo desenvolvedor como pelo usuário do componente, para que sejam realizados os testes de forma sistemática e com qualidade. O desenvolvedor do componente não conhece todos os contextos de utilização do componente e, portanto, o componente pode ter um comportamento em determinados ambientes conforme o esperado e, em outros, não. O usuário sofre com a falta de documentação adequada sobre o componente, agravada muitas vezes, pelo fato de o código fonte não estar disponível. A troca de informações entre eles, desenvolvedor e usuário do componente, é fundamental para o sucesso da utilização do componente, mas nem sempre isso ocorre.

A atividade de teste é importante em todo processo de desenvolvimento de software, pois tem como objetivo principal revelar defeitos em produtos que estão sendo testados. Apesar dos vários aspectos do desenvolvimento de software baseado em componente, Vinzenzi et al (2005) afirmam que as técnicas tradicionais de teste podem ser utilizadas nos testes de software baseado em componentes. Dentre as técnicas tradicionais de testes destaca-se neste trabalho a técnica estrutural, que tem como objetivo garantir que determinadas estruturas do código da implementação sejam exercitadas. Tais estruturas, como comandos, desvios ou pontos do programa onde as variáveis são utilizadas, constituem os requisitos de teste a serem satisfeitos pelos casos de teste. A porcentagem de tais requisitos executados – ou cobertos – pelos casos de teste, representa a cobertura desse conjunto em relação ao teste estrutural.

Neste contexto, foi desenvolvida a Ferramenta FATEsC – Ferramenta de Apoio ao Teste Estrutural de Componentes – com o objetivo de contribuir com a proposição de uma estratégia para o teste de componentes utilizando a técnica de teste estrutural. Foi implementada como uma extensão da ferramenta JaBUTi (VINZENZI et al, 2006) para gerar e disponibilizar metadados encapsulados com o código do componente, fornecendo dados sobre a cobertura dos testes realizados pelo desenvolvedor do componente ao usuário. A ferramenta JaBUTi permite a automação de determinados processos da fase de teste estrutural para programas escritos na linguagem de programação Java, necessitando apenas do bytecode para suas atividades. Essa característica deve permitir que os dados de cobertura de teste do componente, alcançados durante o teste efetuado pelo desenvolvedor, possam ser reutilizados pelo usuário do componente.

Na próxima seção é apresentada a Ferramenta FATEsC e o processo de sua utilização através de um exemplo. A Seção 3 contém as considerações finais deste artigo e propostas de trabalhos futuros.

## **2. FATEsC – Ferramenta de Apoio ao Teste Estrutural de Componentes**

FATEsC é uma ferramenta *DeskTop*, desenvolvida na linguagem de programação Java e é composta por duas partes: uma para ser utilizada pelo desenvolvedor do componente e outra para ser utilizada pelo usuário do componente. Basicamente tem as mesmas funcionalidades, exceto pela comparação das coberturas obtidas pelos casos de teste do desenvolvedor e do usuário do componente, que aparece apenas na versão do usuário.

Na FATEsC, perspectiva do desenvolvedor do componente, é possível criar, manipular e gerenciar casos de teste para cada método público do código, além de definir para cada caso de teste uma descrição informal de suas características. Após definidos os casos de teste, a ferramenta auxilia o desenvolvedor a avaliar a cobertura de cada um deles usando a ferramenta JaBUTi e associa essa informação a cada um dos métodos públicos testados. Essa informação é anexada ao componente (arquivo jar) por meio de metadados no formato XML, tornando-se disponível aos usuários do componente.

Na perspectiva do usuário do componente, é possível criar e avaliar a adequação dos casos de teste e comparar medidas de cobertura obtidas, com as fornecidas nos metadados. Por exemplo, suponha-se um método **m**, da interface pública do componente. Se na execução dos casos de teste do usuário obteve-se a mesma cobertura, em relação ao código de **m** que os casos de teste do desenvolvedor, isso pode significar que realmente os casos de teste estão adequados. Caso contrário pode-se analisar os casos de testes descritos nos metadados e verificar se há a necessidade de criar novos casos de teste ou se para a aplicação em questão, não é possível obter a mesma cobertura. É importante notar que para a aplicação desta estratégia é essencial que o usuário possa avaliar a cobertura obtida sobre o código do componente, sem que o programa fonte lhe esteja disponível. Por essa razão optou-se pela utilização da ferramenta JaBUTi, que utiliza apenas o código objeto (bytecode Java) para realizar a análise de cobertura.

### 2.1. Processo de utilização da Ferramenta

Para melhor compreensão da proposta desse trabalho e a utilização da ferramenta propõe-se um exemplo simples considerando como componente a classe **Fat** que faz o cálculo do fatorial de um número e uma aplicação, que utiliza o componente, a classe **Agrupamentos** que faz o cálculo de combinação simples. As Figuras 1 e 2 mostram o código fonte para o exemplo proposto.

```
public class Fat
{
    /* calcula o fatorial de um
    número inteiro.
    O valor passado deve ser >= 0
    e <= 20.
    O valor retornado é um long
    */
    public long fat(int x)
    {
        if ( x < 0 || x > 20 )
            return -1;
        if ( x == 0 )
            return 1;
        if ( x <= 2 )
            return x;
        long s = 1;
        while ( x > 1 )
        {
            s *= x;
            x--;
        }
    }
}
```

Figura 1. Código fonte do componente proposto como exemplo.

```

public class Agrupamentos {

    /* Calcula o número de combinações de k
    elementos n a n.
    Os valores passados devem ser >=1 e de
    deve ser >= n
    */

    public long combinacao(int k, int n)
    {
        Fat f = new Fat();
        if (k < 1 || n < 1 || k < n )
            throw new IllegalArgumentException( "Argumento invalido");
        long s = f.fat(k);
        s /= f.fat(n);
        s /= f.fat(k-n);
        return s;
    }
}
    
```

Figura 2. Código fonte da aplicação que utiliza o componente (Fat) proposto como exemplo.

A partir de um arquivo jar contendo o bytecode do componente, cuja estrutura hierárquica é apresentada na parte esquerda (1) da tela da Figura 3, o desenvolvedor do componente pode criar casos de teste para cada interface pública do componente. Os casos de teste ficam listados na parte direita da tela (3), podendo a qualquer tempo ser selecionado e o código fonte de cada caso de teste, no formato JUnit, bem como a descrição deles podem ser vistos e editados na parte central da tela (2).

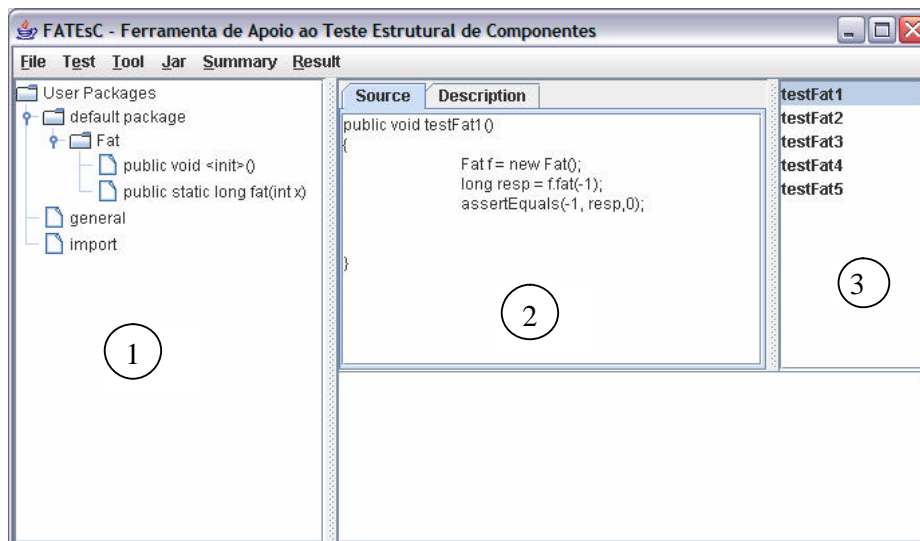


Figura 3. Tela principal da FATEsC: (1) hierarquia do componente, (3) lista de casos de teste e (2) código fonte do caso de teste testFat1.

Após a geração dos casos de teste, a FATEsC disponibiliza a funcionalidade de juntar todos os casos de testes em um arquivo único e compilá-lo. Os casos de teste



devem estar no formato JUnit (framework para desenvolvimento de teste de unidade em Java) (MASSOL e HUSTED, 2005) e nesse momento pode-se executá-los na Ferramenta JaBUTi.

As informações de cobertura dos testes, realizados na Ferramenta JaBUTi, são capturadas pela Ferramenta FATEsC e disponibilizadas no formato XML, ou seja, para cada caso de teste utilizado para testar uma determinada interface pública do componente, obtém-se os requisitos cobertos separados por critério de teste. Um requisito coberto pode ser um comando, um desvio, uma associação definição-uso ou outro elemento estrutural, definido pelo critério de teste que está sendo usado pelo desenvolvedor.

Os dados de cobertura dos casos de testes e as descrições dos casos de testes são disponibilizados junto ao componente, o que significa adicionar metadados XML gerados pela FATEsC ao arquivo contendo o bytecode do componente no formato .jar, que será usado pelo usuário do componente.

O usuário do componente, por sua vez, também utiliza a FATEsC para criar os casos de testes para a sua aplicação que utiliza o componente, o processo é o mesmo percorrido pelo desenvolvedor do componente, conforme Figura 4. Após a obtenção dos dados de cobertura dos seus casos de testes executados na Ferramenta JaBUTi, o usuário do componente, poderá fazer a análise da adequação de seus casos de testes, comparando-os com os dados de cobertura obtidos pelo desenvolvedor do componente.

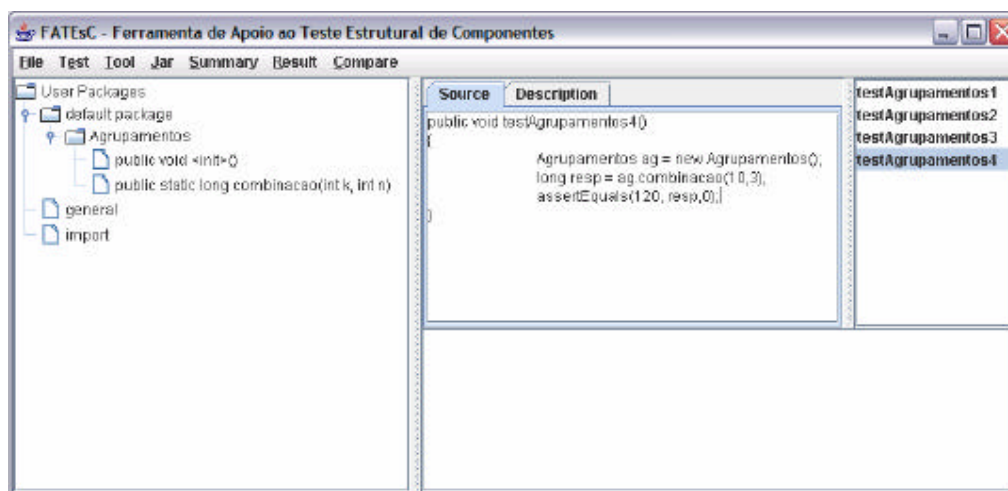
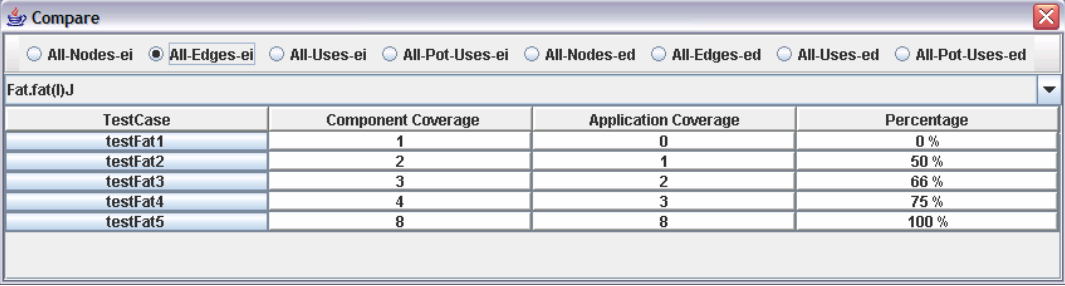


Figura 4. Tela principal da FATEsC: hierarquia da aplicação, lista de casos de teste e código fonte do caso de teste testAgrupamentos4.

O usuário deverá selecionar o critério e a interface pública do componente para o qual os resultados serão comparados, conforme apresentado na Figura 5. Na primeira coluna são listados os casos de testes gerados pelo desenvolvedor do componente. Na segunda coluna, são apresentados os dados de cobertura do componente, ou seja, quantidade de requisitos cobertos pelo caso de teste. Na terceira coluna são apresentadas as quantidades de requisitos cobertos pelos casos de teste do usuário em relação à cobertura obtida pelo desenvolvedor do componente e na quarta e última coluna o cálculo da porcentagem de cobertura obtida pelo usuário do componente em relação à cobertura obtida pelo desenvolvedor do componente. No exemplo, o caso de teste **testFat5** cobriu oito requisitos e os casos de teste do usuário também cobriram os

mesmos requisitos, portanto, a cobertura foi de 100%. Os requisitos cobertos, neste exemplo, referem-se a desvios do programa, visto que o critério escolhido é Todos-arcos.

Já nos outros casos de testes, a cobertura não foi a mesma, ou seja, os casos de testes do usuário do componente, não conseguiram cobrir os mesmos requisitos cobertos pelos casos de testes do desenvolvedor do componente, nesse caso é necessário que se faça uma análise com o objetivo de verificar se os casos de teste do usuário não estão adequados, ou se os casos de teste do componente, não podem ser reproduzidos no contexto da aplicação.



TestCase	Component Coverage	Application Coverage	Percentage
testFat1	1	0	0 %
testFat2	2	1	50 %
testFat3	3	2	66 %
testFat4	4	3	75 %
testFat5	8	8	100 %

Figura 5. Apresentação dos dados de cobertura obtidos pelo usuário e pelo desenvolvedor do componente.

Para ajudar o usuário do componente em sua avaliação, é possível visualizar as descrições dos casos de testes disponibilizadas pelo desenvolvedor, bastando para isso selecionar o caso de teste desejado. Para o caso de teste **testFat1**, conforme apresentado na Figura 6, a descrição diz que foi passado um argumento negativo para o método **fat** e como a aplicação do usuário não faz acesso à interface pública do componente para parâmetros  $k < 1$ ,  $n < 1$  e  $k < n$  conclui-se que esse caso de teste não pode ser reproduzido para a aplicação. Nesse caso marca-se “infeasible” na tela de apresentação da descrição do caso de teste e o caso de teste é apresentado com uma cor diferente (cinza) representando a avaliação feita, conforme mostrado na Figura 7.

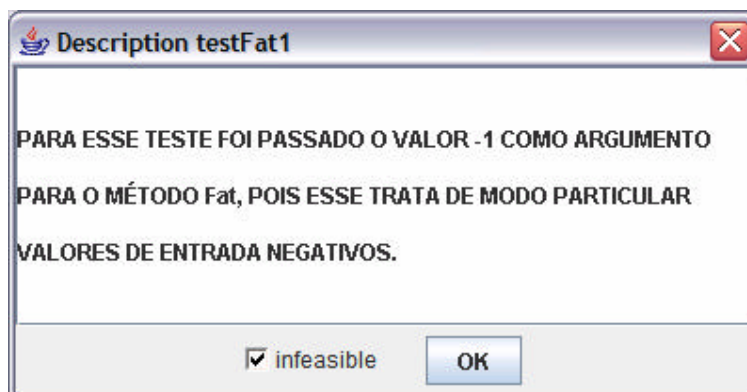


Figura 6. Apresentação da descrição do caso de teste testFat1.

TestCase	Component Coverage	Application Coverage	Percentage
testFat1	1	0	0 %
testFat2	2	1	50 %
testFat3	3	2	66 %
testFat4	4	3	75 %
testFat5	8	8	100 %

Figura 7. Aparência de um caso de teste (testFat1) que não poderá ser reproduzido na aplicação do usuário do componente.

Ao analisar o caso de teste **testFat4**, verificando a descrição do caso de teste apresentado na Figura 8 o usuário do componente perceberá que poderá alcançar a mesma cobertura que o caso de teste do componente, necessitando apenas criar um novo caso de teste que passe como parâmetros  $k = 10$  e  $n = 2$ .

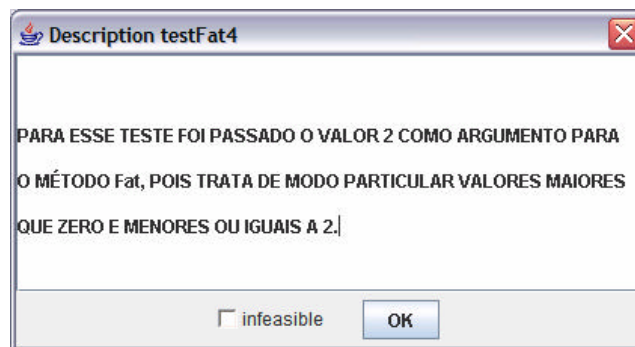


Figura 8. Apresentação da descrição do caso de teste testFat4.

Ao analisar o caso de teste **testFat2**, verificando a descrição do caso de teste apresentado na Figura 9, o usuário do componente perceberá que o cálculo do fatorial de 25 não é -1, então poderá se lembrar que o componente calcula o fatorial até o número 20 e que esta limitação deverá ser tratada em sua aplicação, isso mostra como a estratégia adotada por esse trabalho ajuda a revelar defeitos. Assim, o usuário do componente deveria continuar suas análises até que as coberturas se iguallassem ou se concluísse que alguns casos de testes não poderiam ser reproduzidos para a aplicação.

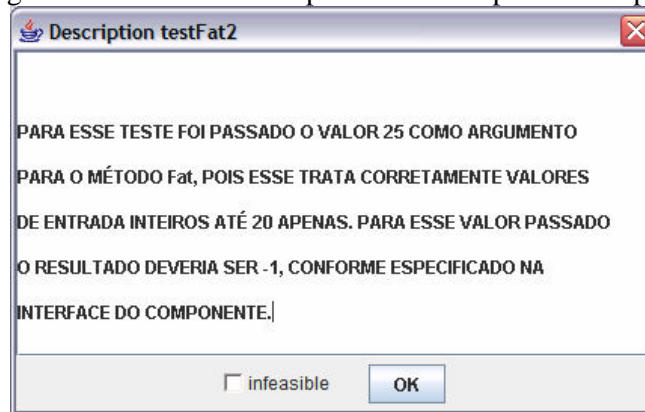


Figura 9. Apresentação da descrição do caso de teste testFat2.

### 3. Conclusão

Apesar de existirem algumas propostas na literatura abordando formas de solucionar efetivamente o problema da falta de informação, poucas tratam objetivamente de quais seriam essas informações e como as mesmas poderiam ser utilizadas. Por meio da utilização conjunta da Ferramenta FATEsC e JaBUTi o desenvolvedor do componente pode gerar casos de teste e executá-los, encapsulando ao componente os dados de coberturas obtidos, bem como as descrições dos casos de teste. O usuário do componente por sua vez, também pode gerar casos de teste para sua aplicação que utiliza o componente, executá-los e fazer uma avaliação da adequação dos casos de teste tendo como base para comparação os dados da cobertura dos testes obtidos pelo desenvolvedor do componente. Esta é uma forma de suprir a falta de informação, beneficiando o usuário do componente ao realizar a atividade de teste em sua aplicação.

Como forma de evolução desse trabalho é importante a condução de estudos de casos em ambientes de desenvolvimento de software baseado em componentes para avaliar a ferramenta, identificado os benefícios obtidos e possíveis adequações necessárias.

### Referências

- BEYDEDA, S. and GRUHN, V. (2003) "State of the art in testing components", In: International Conference on Quality Software (QSIC), IEE Computer Society Press.
- MASSOL, V. and HUSTED, T. (2005) "JUnit Em Ação". Editora Ciência Moderna.
- VINCENZI, A., DELAMARO, M.E., WONG, E., MALDONADO, J.C. and SPOTO, E.S (2005) "Software baseado em componentes: uma revisão sobre teste". In: Desenvolvimento baseado em components: conceitos e técnicas. Editora Ciência Moderna.
- VINCENZI, A., DELAMARO, M.E., WONG, E. and MALDONADO, J.C. (2006) "Establishing Structural Testing Criteria for Java Bytecode". *Software, Practice & Experience*, v. 36, p. 1513-1541.

# Uma Ferramenta para Configuração e Implantação de Sistemas Distribuídos de Tempo-Real Baseados em Componentes

Sandro S. Andrade, Aristoteles M. Silva e Cleber N. Ramos

Faculdade Ruy Barbosa  
Rua Theodomiro Batista, 422 - Rio Vermelho - Cep: 41.940-320  
Fone/Fax : +55 (71) 3205-1733 - Salvador - Bahia - Brasil

{sandros, clebernr, aristotelesms}@frb.br

**Abstract.** *Meeting demands related to distribution, flexibility, reusability, and interoperability in real-time systems has been the goal of many current researches, some of them devoted to the use of distributed component technology as an important mechanism for complexity management and temporal predictability. This paper presents the design and implementation of a tool aiming the visual configuration and deployment of component-based distributed systems built atop of CIAO: a recent implementation of the CORBA Component Model (CCM) standard, with real-time extensions. The proposed tool provides visual mechanisms for implementing and importing CIAO components, interconnecting components into an assembly, configuring of real-time parameters, and deploying of the component-based system in different network nodes.*

**Resumo.** *O atendimento de demandas relacionadas a distribuição, flexibilidade, reutilização e interoperabilidade em sistemas de tempo-real tem sido o foco de diversas pesquisas recentes, algumas direcionadas ao uso de componentes distribuídos como tecnologia para gerenciamento da complexidade e para previsibilidade temporal. Este artigo apresenta o projeto e implementação de uma ferramenta para configuração e implantação visual de sistemas baseados no CIAO: uma implementação recente do CORBA Component Model (CCM), com extensões para tempo-real. A ferramenta disponibiliza mecanismos gráficos para implementação e importação de componentes CIAO, combinação visual entre componentes, configuração de requisitos temporais e implantação remota da montagem em diferentes nós de uma rede.*

## 1. Introdução

Nos últimos anos, a crescente evolução das tecnologias de processamento e comunicação de dados tem demandado novos requisitos para sistemas computacionais. A utilização de plataformas flexíveis, escaláveis, reutilizáveis e interoperáveis são características atualmente desejadas e a complexidade gerada pela introdução de tais requisitos requer a adoção de metodologias, mecanismos e ferramentas que auxiliem o desenvolvimento controlado e produtivo de tais sistemas.

A utilização de componentes distribuídos de software [Gimenes and Huzita 2005, Heineman and Councill 2001] tem sido uma prática promissora para a obtenção de aplicações e plataformas flexíveis, reutilizáveis e escaláveis. Ao possibilitar a construção

de sistemas através da combinação facilitada de peças de software, a tecnologia de componentes disponibiliza uma infra-estrutura adequada ao desenvolvimento produtivo de plataformas, tais como os *frameworks* de aplicação, e de aplicações em diversos domínios.

Em particular, os sistemas distribuídos de tempo-real têm sido o foco de diversas pesquisas interessadas no estudo da viabilidade de utilização, em tais sistemas, de soluções de *middleware* para objetos e componentes distribuídos. Tecnologias tais como o TAO (*The ACE ORB*) [Schmidt et al. 1998] e o CIAO (*Component-Integrated ACE ORB*) [N. Wang and Subramonian 2004] representam esforços importantes no sentido de adequar as tecnologias atuais da engenharia de software para uso em sistemas de tempo-real.

Em adição à inerente complexidade introduzida pelas demandas atuais, implementações de determinados modelos de componentes distribuídos tendem a inserir artifícios relacionados diretamente aos serviços e funcionalidades disponibilizados pelo *middleware*, tais como arquivos de definição de interfaces, métodos de *callback* e descritores XML de configuração e implantação. Ao mesmo tempo em que constituem ferramentas indispensáveis para o desenvolvimento de sistemas distribuídos modernos, as soluções de *middleware* para componentes ainda requer, do desenvolvedor, um conhecimento especializado sobre o modelo implementado e sobre a arquitetura utilizada.

Este artigo apresenta o projeto e implementação do ATOME: uma ferramenta para configuração e implantação de sistemas distribuídos de tempo-real baseados em componentes. O ATOME possibilita o desenvolvimento produtivo de sistemas distribuídos de tempo-real baseados no *CORBA Component Model*<sup>1</sup> (CCM) [OMG 2001], através da disponibilização de mecanismos para: geração automática dos artefatos requeridos pelo CIAO para criação e implantação de componentes, conexão e configuração visual de componentes, criação do mapa de nós do ambiente de implantação, configuração dos atributos temporais de instâncias e implantação distribuída da montagem.

Dentre os objetivos do ATOME destacam-se: i) proporcionar uma transparência ao desenvolvedor em relação aos artifícios requeridos pelo CCM/CIAO, possibilitando o foco no negócio da aplicação, ii) possibilitar a conexão e configuração visual dos componentes de uma montagem, incluindo restrições temporais e iii) automatizar o processo de implantação distribuída do sistema.

O restante do artigo está organizado como segue. A seção 2 apresenta os fundamentos e tecnologias que motivaram a construção da ferramenta proposta. A seção 3 apresenta a arquitetura e o projeto do ATOME, suas funcionalidades e aspectos relacionados à sua implementação, enquanto a seção 4 apresenta os trabalhos correlatos a este projeto. Finalmente, a seção 5 apresenta as conclusões e trabalhos futuros.

## 2. Componentes Distribuídos para Tempo-Real

A aplicação de tecnologias da engenharia de software em sistemas de tempo-real tem sido o foco de pesquisas recentes [Andrade and Macêdo 2007, A. Tesanovic and Norstom 2004]. Em particular, soluções para componentes distribuídos de tempo-real [Hatcliff et al. , N. Wang and Subramonian 2004] têm sido experimentadas em cenários tais como sistemas industriais de supervisão e controle, sistemas robóticos

---

<sup>1</sup>Modelo de componentes implementado pelo CIAO. Padronização para componentes distribuídos criada pelo Object Management Group (OMG), em 2001.

e sistemas embarcados. Esta seção apresenta brevemente o *CORBA Component Model* (CCM) e as extensões para tempo-real implementadas pelo *Component-Integrated ACE ORB* (CIAO).

## 2.1. CORBA Component Model (CCM)

O *Object Management Group* (OMG) disponibilizou em junho de 2001 a versão 3 do CORBA, contemplando um modelo e uma especificação de componentes denominada *CORBA Component Model* (CCM) [OMG 2001]. O CCM define um modelo de componentes distribuídos, mecanismos para conexão entre componentes e padronizações para implantação, configuração e manutenção de componentes.

Um componente CCM é uma unidade de implantação e conexão que implementa funcionalidades reutilizáveis do sistema computacional em questão. Essas funcionalidades são disponibilizadas através de métodos pertencentes à(s) interface(s) suportada(s) pelo componente. Uma interface suportada é uma interface CORBA da qual o componente realiza uma implementação, definindo os métodos que podem ser invocados por aplicações clientes daquele componente.

As conexões entre componentes são realizadas através de mecanismos denominados *ports*. O CCM define quatro tipos de *ports*: i) facetas (*facets*) - representam uma funcionalidade provida pelo componente, ii) receptáculos (*receptacles*) - representam uma funcionalidade requerida pelo componente e, em conjunto com as facetas, constituem um mecanismo para conexão síncrona (bloqueante) de componentes, iii) produtor de eventos (*event sources*) - representam geradores de eventos e iv) consumidores de eventos (*event sinks*) - representam receptores de eventos e, em conjunto com os produtores de eventos, constituem um mecanismo para conexão assíncrona (não-bloqueante) de componentes.

Todas as operações de conexão de facetas e receptáculos, bem como de produtores e depósitos de eventos, são realizadas pelo *middleware*, a partir de indicações nos arquivos XML descritores de implantação. Além dos *ports*, o CCM define pontos de configuração do componente denominados atributos. O valor do atributo é informado, via arquivo descritor, no momento da implantação do componente. Um atributo serve para adequar o funcionamento do componente a uma situação particular.

## 2.2. Component-Integrated ACE ORB (CIAO) e Tempo-Real

O CIAO (*Component-Integrated ACE ORB*) [N. Wang and Subramonian 2004] é uma implementação da especificação CCM desenvolvida pelo DOC Group<sup>2</sup> e baseia-se na utilização de duas tecnologias bastante consolidadas na área de *middleware* para tempo-real: o ACE (*ADAPTIVE Communication Environment*) [Schmidt and Huston 2002] - um *framework* para desenvolvimento de sistemas distribuídos de alto desempenho - e o TAO (*The ACE ORB*) [Schmidt et al. 1998] - uma implementação da especificação CORBA 2.x (baseada em objetos distribuídos), com extensões para tempo-real.

Enquanto o CCM destaca-se dos demais modelos devido aos mecanismos expressivos para conexão e configuração de componentes e serviços para implantação disponibilizados, o CIAO sobressai-se em relação às outras implementações do CCM em função das seguintes características: i) implementação de alto desempenho (devido às

---

<sup>2</sup><http://www.cs.wustl.edu/schmidt/TAO.html>

otimizações realizadas em todos os participantes da arquitetura CORBA), ii) execução previsível de componentes (através da configuração, via XML, dos parâmetros definidos pelo RT-CORBA [(OMG) 2003] e da implementação do *container* de tempo-real), iii) serviço para configuração e implantação (todo o processo de implantação e conexão de componentes em diferentes nós da rede, bem como a gerência do repositório de componentes, são realizados pelo DANCE - *Deployment And Configuration Engine*) e iv) mecanismos para reconfiguração dinâmica (adição e remoção de instâncias e conexões são realizadas pelo ReDaC - *Redeployment and Reconfiguration* - sem requerer a interrupção do sistema).

O RT-CORBA [(OMG) 2003] é uma especificação, baseada no CORBA 2.x, que define extensões para a implementação de objetos distribuídos para sistemas de tempo-real com prioridades fixas. As invocações com prioridades no RT-CORBA seguem um dos modelos especificados: *client-propagated* e *server-declared*. No modelo *client-propagated*, a prioridade é informada pelo cliente no momento da invocação e o servidor ajusta o ambiente para executar a requisição. No modelo *server-declared* as prioridades são pré-definidas pelo servidor no momento da implantação. Ao adquirir uma referência para um componente, o cliente obtém a prioridade declarada pelo servidor e esta não poderá ser alterada.

Com o objetivo de garantir a previsibilidade das invocações, o RT-CORBA define dois modelos de reserva de recursos: o *thread-pool* e o *thread-pool with lanes*. No modelo *thread-pool*, um conjunto de *threads* (*pool*) é previamente criado e os seguintes parâmetros podem ser configurados: número de *threads* estáticas, número de *threads* dinâmicas, prioridade das *threads* e tamanho do *buffer* de requisições. O RT-CORBA gerencia e manipula o *pool* de *threads* com base nessa parametrização, com o objetivo de prover um ambiente de execução previsível. No modelo *thread-pool with lanes*, diversos grupos (*lanes*) de *threads* podem ser criados dentro de um mesmo *pool*. Cada grupo possui seus próprios parâmetros de configuração e pode tomar *threads* "emprestadas" de outros grupos, em situações de escassez de recursos.

O CIAO disponibiliza um mecanismo facilitado para configuração dos parâmetros do RT-CORBA através da utilização de descritores XML, em contraste à forma custosa e propensa a erros do CORBA 2.x, realizada via código-fonte.

### 3. A Ferramenta Proposta

A ferramenta proposta neste trabalho, ATOME, integra as tecnologias oferecidas pelo CIAO e disponibiliza um ambiente fácil e produtivo para o desenvolvimento de sistemas baseados em componentes CCM. A Figura 1 apresenta a arquitetura do ATOME e o seu relacionamento com as tecnologias do CIAO.

O ATOME possui duas macro-funcionalidades importantes: geração de versões iniciais de componentes CCM e configuração e implantação de montagens de tempo-real. Motivado pela considerável complexidade de criação de um componente CIAO (principalmente por desenvolvedores inexperientes), o ATOME disponibiliza recursos gráficos para criação de componentes através da indicação da(s) interface(s) suportada(s), facetas, receptáculos, produtores, consumidores e atributos que o compõem. Para a definição de uma montagem, o ATOME disponibiliza um editor gráfico para conexão e configuração de atributos e parâmetros temporais dos componentes gerados pela ferramenta ou impor-



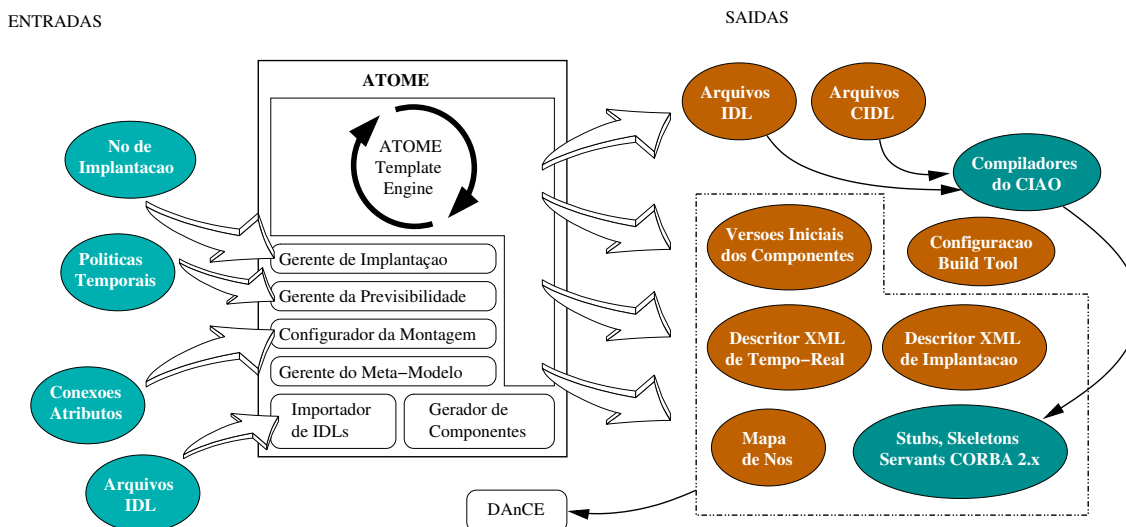


Figura 1. Arquitetura do ATOME e tecnologias auxiliares.

tados através de seus arquivos IDL.

Os principais participantes definidos na arquitetura do ATOME são:

- Gerador de Componentes: responsável pela criação de componentes CCM/CIAO através de indicações visuais realizadas na ferramenta, tais como criação de interfaces, atribuição de interfaces a facetas, criação de receptáculos, criação de atributos, definição de tipos de eventos e criação de produtores e consumidores. Interage com o *ATOME Template Engine*, responsável pela geração dos seguintes artefatos: arquivos IDL e CIDL, versão inicial (obviamente sem lógica de negócio) da implementação do componente e arquivo de configuração da ferramenta de compilação MPC (*Make Project Creator*);
- Importador de IDLs: componentes CCM/CIAO previamente criados podem ser importados para uso em futuras montagens. O importador de IDL realiza o *parsing* do arquivo IDL, informando a estrutura do componente para o Gerente do Meta-Modelo;
- Gerente do Meta-Modelo: responsável pelo armazenamento de meta-informações referentes aos componentes importados/criados e à montagem sendo configurada. Implementa verificações de consistência na montagem, tais como compatibilidade entre *ports* (impede conexões inapropriadas ou entre tipos distintos) e detecção de componentes isolados;
- Configurador da Montagem: interage com o Gerente do Meta-Modelo a partir de requisições de conexão e configuração de componentes realizadas pelo usuário. Armazena o mapa de nós do domínio de implantação, contendo os endereços IP e portas das máquinas que podem hospedar componentes;
- Gerente da Previsibilidade: disponibiliza mecanismos para criação das políticas do RT-CORBA (*thread pools* e modelos de invocação com prioridade). Possibilita a atribuição dessas políticas às instâncias criadas na montagem;
- Gerente de Implantação: responsável pela coleta dos artefatos necessários à implantação distribuída da montagem. Utiliza os serviços subjacentes, disponibilizados pelo DAnCE;

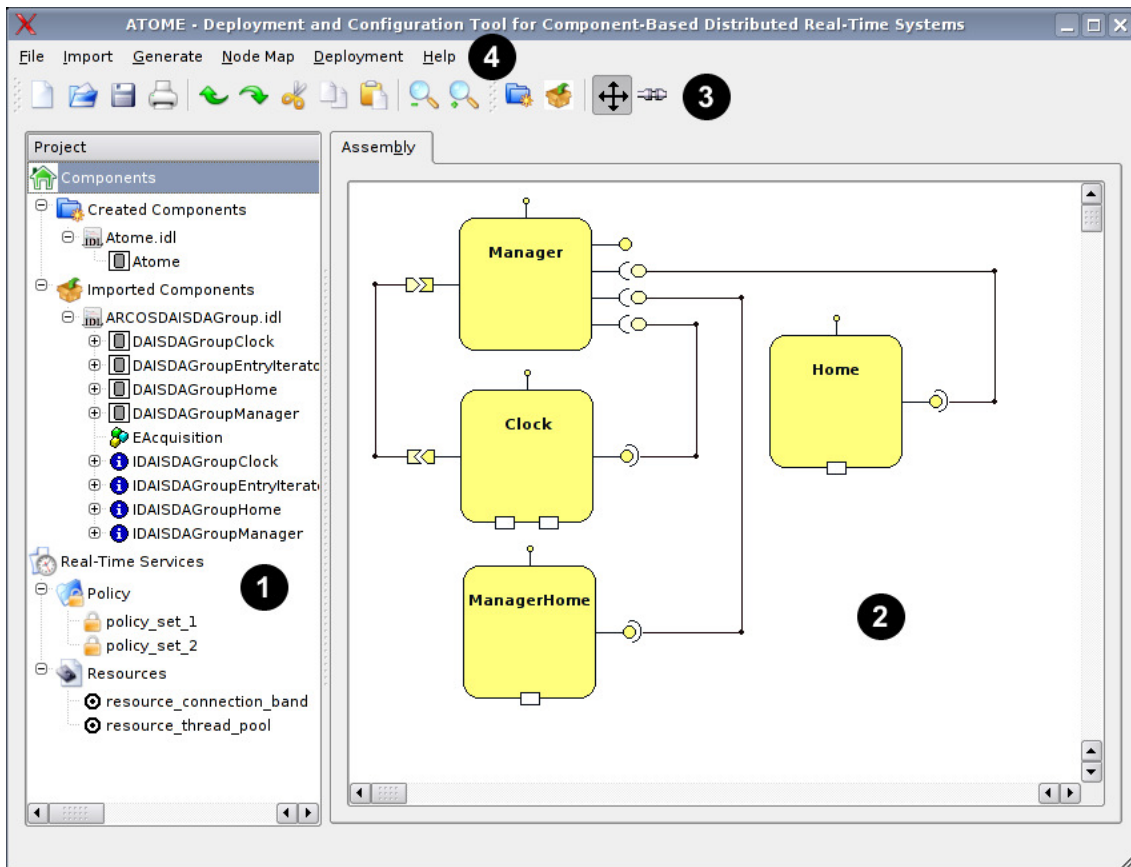


Figura 2. A ferramenta ATOME.

- *ATOME Template Engine*: núcleo central para geração de artefatos. Utiliza arquivos *template* contendo versões parametrizadas de todos os artefatos gerados.

A Figura 2 apresenta a ferramenta ATOME exibindo a construção de uma montagem. A área 1 apresenta a visão dos recursos disponíveis para montagem. A árvore "Components" apresenta os componentes criados pelo usuário através da ferramenta, bem como os componentes importados através dos seus arquivos IDL. A árvore "Real-Time Services" contém os recursos e políticas de tempo-real definidos pelo usuário (*thread-pools*, *lanes* e recursos relacionados à rede de comunicação). Componentes e serviços de tempo-real podem ser facilmente criados através de menus de contexto (botão direito do *mouse*) disponíveis em ramos das árvores apresentadas.

A área 2 apresenta o editor de montagens implementado no ATOME. Nesse editor o usuário pode solicitar a criação de uma nova instância de componente, indicar o nó do domínio no qual a instância será implantada, realizar conexões com outras instâncias criadas, ajustar valores de atributos e atribuir políticas de tempo-real. Com essa abstração, usuários iniciantes na tecnologia CCM/CIAO poderão focar nas questões de negócio da aplicação, deixando para o ATOME a responsabilidade da geração correta dos artefatos necessários. Ressalta-se a alta complexidade e propensão a erros da construção manual dos descritores XML de implantação e de tempo-real. As áreas 3 e 4 apresentam, respectivamente, a barra de tarefas e as operações de menu da ferramenta ATOME.

	Cadena	CoSMIC	ATOME
Plataforma de <i>middleware</i> utilizada	OpenCCM e CIAO	ACE, TAO e CIAO	CIAO
Suporte a tempo-real	Sim	Sim	Sim
Mecanismo para montagem e criação de componentes	Edição de arquivos	UML	Gráfico
Integração com outra ferramenta	Eclipse	Não	Não
Suporte ao processo de implantação	Não	Sim	Sim

Tabela 1. Comparação do ATOME com as demais ferramentas.

### 3.1. Aspectos de Implementação

O ATOME foi projetado e desenvolvido no sistema operacional GNU/Linux, utilizando a linguagem Standard C++ e as tecnologias TrollTech Qt3 e KDevelop. O Qt3 é um *toolkit* para desenvolvimento de aplicações visuais modernas, utilizando a linguagem C++. Caracterizado pela sua excelente portabilidade entre os ambientes GNU/Linux, Microsoft Windows e MAC OS, o Qt3 disponibiliza um conjunto extensivo de classes para *multithreading*, manipulação de XML, multimídia, banco de dados, além dos convencionais recursos para construção de interfaces gráficas.

O KDevelop é um IDE (*Integrated Development Environment*) para desenvolvimento de software livre na plataforma GNU/Linux. O KDevelop atualmente suporta cerca de doze linguagens de programação e possui forte integração com as ferramentas do Qt, constituindo um ambiente produtivo para o desenvolvimento de sistemas modernos. Todos os módulos do ATOME foram integralmente desenvolvidos a partir das API's disponibilizadas pelo Qt facilitando, dessa forma, a sua migração para outras plataformas. O ATOME encontra-se atualmente em processo de migração para o Qt4, o que possibilitará a sua execução no ambiente Microsoft Windows sem a necessidade de adoção da licença proprietária requerida pelo Qt3 neste ambiente.

## 4. Trabalhos Correlatos

Dentre as atuais ferramentas de auxílio ao desenvolvimento de sistemas de tempo-real baseados em componentes pode-se destacar: o CoSMIC (*Component Synthesis using Model Integrated Computing*) [Gokhale et al. 2003] e o Cadena [Hatcliff et al. ]. O CoSMIC reúne uma série de aplicações para especificação, verificação, implementação, configuração e montagem de sistemas de tempo-real distribuídos baseados em componentes e na tecnologia MDA (*Model-Driven Architecture*). O CoSMIC tem como objetivo a geração dos artefatos de um sistema de tempo-real a partir de um modelo UML fornecido pelo desenvolvedor.

O Cadena [Hatcliff et al. ] promove extensões para tempo-real a partir de uma implementação do CCM denominada OpenCCM. Além das extensões para tempo-real, o Cadena disponibiliza uma ferramenta para composição de montagens e geração dos descritores XML utilizados na implantação. A tabela 1 apresenta a comparação entre o ATOME e as demais ferramentas.

## 5. Conclusões e Trabalhos Futuros

Este artigo apresentou o ATOME: uma ferramenta para configuração e implantação visual de sistemas distribuídos de tempo-real baseados em componentes CCM/CIAO. Com a adoção e amadurecimento prático do desenvolvimento de sistemas baseados em componentes, ferramentas de configuração e implantação de montagens se tornam fundamentais para o desenvolvimento produtivo de sistemas. Neste contexto, o papel do montador de software é indispensável para a implantação de metodologias e práticas para o desenvolvimento ágil de sistemas.

Dentre os trabalhos futuros da ferramenta destacam-se: projeto e implementação do módulo de integração com repositórios do DAnCE, suporte a "profiles" com o objetivo de direcionar montagens de sistemas construídos a partir de *frameworks* baseados em componentes, migração para o Qt4 (disponibilizando uma versão *open source* para o ambiente Microsoft Windows) e implementação do módulo de gerência do ambiente distribuído, responsável pelo monitoramento das instâncias em cada nó do domínio. Maiores informações sobre a ferramenta ATOME, bem como obtenção de versões binárias e de código-fonte podem ser encontradas em <http://atome.sourceforge.net>.

## Referências

- A. Tesanovic, D. Nystrom, J. H. and Norstom, C. (2004). Aspects and components in real-time system development: Towards reconfigurable and reusable software. *Journal of Embedded Computing*.
- Andrade, S. and Macêdo, R. (2007). Engineering components for flexible and interoperable real-time distributed supervision and control systems. In *12th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'2007)*. September 25-28, 2007. Patras-Greece.
- Gimenes, I. M. D. S. and Huzita, E. H. M. (2005). Desenvolvimento baseado em componentes: Conceitos e técnicas. *Ciencia Moderna*.
- Gokhale, A., Schmidt, D., Lu, T., Natarajan, B., and Wang, N. (2003). Cosmic: An MDA generative tool for distributed realtime and embedded applications.
- Hatcliff, J., Deng, W., Dwyer, M., Jung, G., and Ranganath, V. Cadena: An integrated development, analysis, and verification environment for component-based systems.
- Heineman, G. T. and Councill, W. T. (2001). Component based software engineering: Putting the pieces together. *Addison-Wesley Professional; 1st edition*.
- N. Wang, C. Gill, D. C. S. and Subramonian, V. (2004). Configuring real-time aspects in component middleware. *International Symposium on Distributed Objects and Applications - DOA 2004*.
- OMG, O. M. G. (2001). *CORBA Component Model*.  
<http://www.omg.org/technology/documents/formal/components.htm>.
- (OMG), O. M. G. (2003). *Real-Time CORBA*.  
[http://www.omg.org/technology/documents/formal/RT\\_dynamic.htm](http://www.omg.org/technology/documents/formal/RT_dynamic.htm).
- Schmidt, D. C. and Huston, S. D. (2002). C++ Network Programming: Mastering complexity using ACE and patterns. *Addison-Wesley Longman*.
- Schmidt, D. C., Levine, D. L., and Mungee, S. (1998). The design of the TAO real-time object request broker. *Computer Communications*, 21(4).